

UNCLASSIFIED

AD NUMBER

ADB129568

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies and their contractors; Critical Technology; MAR 1988. Other requests shall be referred to Air Force Armament Lab., Eglin AFB, FL 32542. This document contains export-controlled technical data.

AUTHORITY

AFSC Wright Lab at Eglin AFB, ltr dtd 13 Feb 1992

THIS PAGE IS UNCLASSIFIED

AFATL-TR-88-62, VOL I

Common Ada Missile Packages—Phase 2 (CAMP-2)

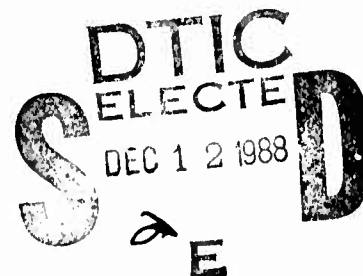
Volume I. CAMP Parts and Parts Composition System

D McNicholl
S Cohen
C Palmer, et al.

AD-B129 568

McDONNELL DOUGLAS ASTRONAUTICS COMPANY
P O BOX 516
ST LOUIS, MO 63166

NOVEMBER 1988



FINAL REPORT FOR PERIOD SEPTEMBER 1985 – MARCH 1988

CRITICAL TECHNOLOGY

Distribution authorized to U.S. Government agencies and their contractors only;
~~this report documents test and evaluation~~; distribution limitation applied March 1988.
Other requests for this document must be referred to the Air Force Armament
Laboratory (FXG) Eglin Air Force Base, Florida 32542-5434.

DESTRUCTION NOTICE – For classified documents, follow the procedures
in DoD 5220.22 – M, Industrial Security Manual, Section II – 19 or DoD 5200.1 – R,
Information Security Program Regulation, Chapter IX. For unclassified, limited
documents, destroy by any method that will prevent disclosure of contents or
reconstruction of the document.

AIR FORCE ARMAMENT LABORATORY

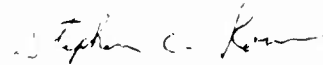
Air Force Systems Command ■ United States Air Force ■ Eglin Air Force Base, Florida

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise as in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



STEPHEN C. KORN
Chief, Aeromechanics Division

Even though this report may contain special release rights held by the controlling office, please do not request copies from the Air Force Armament Laboratory. If you qualify as a recipient, release approval will be obtained from the originating activity by DTIC. Address your request for additional copies to:

Defense Technical Information Center
Cameron Station
Alexandria, VA 22304-6145

If your address has changed, if you wish to be removed from our mailing list, or if your organization no longer employs the addressee, please notify AFATL/FXG, Eglin AFB, FL 32542-5434, to help us maintain a current mailing list.

Do not return copies of this report unless contractual obligations or notice on a specific document requires that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS CRITICAL TECHNOLOGY	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Distribution authorized to U.S. Government Agencies and their contractors; this report contains test and evaluation ; (OVER)	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		5. MONITORING ORGANIZATION REPORT NUMBER(S) AFATL-TR-88-62, Volume I	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		7a. NAME OF MONITORING ORGANIZATION Aeromechanics Division Guidance and Control Branch	
6a. NAME OF PERFORMING ORGANIZATION McDonnell Douglas Astronautics Company	6b. OFFICE SYMBOL (if applicable)	7b. ADDRESS (City, State, and ZIP Code) Air Force Armament Laboratory Eglin Air Force Base, Florida 32542-5434	
6c. ADDRESS (City, State, and ZIP Code) P.O. Box 516 St Louis MO 63166		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F08635-86-C-0025	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION STARS Joint Program Office	8b. OFFICE SYMBOL (if applicable)	10. SOURCE OF FUNDING NUMBERS	
8c. ADDRESS (City, State, and ZIP Code) Room 3D139 (1211 Fern St) The Pentagon Washington DC 20301-3081		PROGRAM ELEMENT NO. 637560	PROJECT NO. 921D
		TASK NO. GT	WORK UNIT ACCESSION NO. 02
11. TITLE (Include Security Classification) Common Ada Missile Packages-Phase 2 (CAMP-2), Volume I: CAMP Parts and Parts Composition System			
12. PERSONAL AUTHOR(S) D.G. McNicholl, S.G. Gohen, C. Palmer, J.F. Mason, C.S. Herr, and J.H. Lindley			
13a. TYPE OF REPORT Final	13b. TIME COVERED FROM Sep 85 to Mar 88	14. DATE OF REPORT (Year, Month, Day) November 1988	15. PAGE COUNT 180
16. SUPPLEMENTARY NOTATION Availability of this report is specified on verso of front cover. (OVER)			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
		Reusable Software, Missile Software, Software Generators, Ada parts, Composition, Systems, Software Parts	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The CAMP project, primarily funded by the STARS Joint Program Office, sponsored by the Air Force Armament Laboratory, and performed by McDonnell Douglas, has taken a pragmatic approach to demonstrating the feasibility and utility of the concept of software reuse for real-time embedded missile systems. CAMP products include: 452 operational flight software parts in Ada for tactical missiles, and a prototype parts engineering system to support parts identification, cataloging and construction. In order to demonstrate the value of the reuse concept, a missile subsystem was built using the CAMP parts. Results indicate a significant increase in software productivity when developing systems using parts, Ada, modern software engineering practice, robust software tools, and knowledgeable software engineers. This report is documented in three volumes: Volume I - CAMP Parts and Parts Composition System, Volume II - 11th Missile Demonstration, and Volume III - CAMP Armonics Benchmarks.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Christine M. Anderson		22b. TELEPHONE (Include Area Code) (904) 882-2961	22c. OFFICE SYMBOL AFATL/FXG

UNCLASSIFIED

3. DISTRIBUTION/AVAILABILITY OF REPORT (CONCLUDED)

distribution limitation applied March 1988. Other requests for this document must be referred to the Air Force Armament Laboratory (FXG), Eglin Air Force Base, Florida 32542-5434.

16. SUPPLEMENTARY NOTATION (CONCLUDED)

TRADEMARKS

The following table lists the trademarks used throughout this document:

TRADEMARK	TRADEMARK OF
ACT	Advanced Computer Techniques
ART	Inference Corporation
ART Studio	Inference Corporation
CMS	Digital Equipment Corporation
DEC	Digital Equipment Corporation
Mikros	Mikros, Inc.
Oracle	Oracle Corporation
Scribe	Scribe Systems
Symbolics	Symbolics, Inc.
Symbolics 3620	Symbolics, Inc.
TLD	TLD Systems Ltd
VAX	Digital Equipment Corporation
VMS	Digital Equipment Corporation

UNCLASSIFIED

PREFACE

This report describes the work performed, the results obtained, and the conclusions reached during the Common Ada Missile Packages Phase-2 (CAMP-2) contract (F08635-86-C-0025). This work was performed by the Software and Information Systems Department of the McDonnell Douglas Astronautics Company, St. Louis, Missouri (MDAC-STL), and was sponsored by the United States Air Force Armament Laboratory (FXG) at Eglin Air Force Base, Florida. This contract was performed between September 1985 and March 1988.

The MDAC-STL CAMP program manager was:

Dr. Daniel G. McNicholl
Technology Branch
Software and Information Systems Department
McDonnell Douglas Astronautics Company
P.O. Box 516
St. Louis, Missouri 63166

The AFATL CAMP program manager was:

Christine M. Anderson
Guidance and Control Branch
Aeromechanics Division
Air Force Armament Laboratory
Eglin Air Force Base, Florida 32542-5434

This report consists of three volumes. Volume I contains information on the development of the CAMP parts and the Parts Composition System. Volume II contains the results of the 11th Missile Application development. Volume III contains the results of the CAMP Armonics Benchmarks Suite development.

Commercial hardware and software products mentioned in this report are sometimes identified by manufacturer or brand name. Such mention is necessary for an understanding of the R & D effort, but does not constitute endorsement of these items by the U.S. Government.

ACKNOWLEDGEMENT

Special thanks to the Armament Division Deputy for Armament Control Office; to the Software Technology for Adaptable, Reliable Systems (STARS) Joint Program Office; to the Ada Joint Program Office (AJPO); and to the Air Force Electronic Systems Division, Computer Resource Management Technology Program Office for their support of this project.



Accession For	
NTIS GRA&I	<input type="checkbox"/>
DTIC TAB	<input checked="" type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
C-2	

TRADEMARKS

The following table lists the trademarks used throughout this document:

TRADEMARK	TRADEMARK OF
ACT	Advanced Computer Techniques
ART	Inference Corporation
ART Studio	Inference Corporation
CMS	Digital Equipment Corporation
DEC	Digital Equipment Corporation
Mikros	Mikros, Inc.
Oracle	Oracle Corporation
Scribe	Scribe Systems
Symbolics	Symbolics, Inc.
Symbolics 3620	Symbolics, Inc.
TLD	TLD Systems Ltd
VAX	Digital Equipment Corporation
VMS	Digital Equipment Corporation

EXECUTIVE SUMMARY

The overall objective of the *Common Ada Missile Packages* (CAMP) program has been to demonstrate the feasibility and value of reusable Ada software parts in DoD mission-critical, real-time, embedded (RTE) applications. As the name of the program implies, the domain chosen for this demonstration was missile operational flight software. Software applications within this domain are typically constrained in terms of memory and timing, and involve a great deal of direct hardware control. As such, if reusable Ada parts could be shown to be suitable for these applications, they would be suitable for use in most other RTE applications.

CAMP is a multi-year research program which has been sponsored by the Air Force Armament Laboratory at Eglin Air Force Base, and performed by the McDonnell Douglas Astronautics Company - St. Louis (MDAC-STL). The program was partially funded by the Air Force Armament Division, the DoD Software Technology for Adaptable, Reliable Systems (STARS) Program Office, the Air Force Electronic Systems Division, and the Ada Joint Program Office (AJPO).

The ability to reuse pre-existing software components to build new applications has been identified by most software engineering organizations as a key element in their plans to reduce software development costs and schedules. However, prior to the mid-1980s, few organizations have been able to achieve wide-spread, systematic reuse of software. One of the major barriers to software reuse has been that traditional programming languages were not designed with reuse in mind. With the adoption of Ada as the DoD standard computer programming language for mission-critical computer systems, many DoD software engineers believed that meaningful, systematic software reuse was feasible for the first time.

Ada promotes reuse of software in two ways. First, it is a highly transportable language. Software written in Ada can be moved from one type of computer to another relatively easily. This property of Ada facilitates reusing software between applications hosted on different computers. Second, specific features were built into the Ada language to allow a user to construct powerful software components that are transportable between applications.

When Ada was released, many software managers and engineers quickly saw the advantages in developing standard reusable parts or components that could be used across a spectrum of applications and computer types. Their vision was to treat software engineering the same way other engineering disciplines are treated — build customized components only when needed and reuse standard parts whenever possible.

However, one very important portion of the software engineering community expressed a great deal of skepticism with the concept of reusable software — software engineers building RTE applications. These applications are characterized by severe memory and timing constraints and the need to have direct control over the computer and its attached equipment. The RTE software community needed to be convinced that reusable Ada parts could be developed which were both sufficiently effective and efficient for the types of applications they needed to build. In the rush to exploit the potential of Ada for reusable software in general, no one was addressing these RTE applications. There was a very good reason for this — developing reusable software parts for RTE applications is much more difficult than building reusable software parts for non-RTE applications.

Given the pervasiveness of RTE applications within the DoD, there was an urgent need to examine whether reusable Ada parts could be built which were suitable for use in RTE applications. In 1984, the U.S. Air Force addressed this need by initiating the CAMP program.

The first phase of the CAMP program, the CAMP-1 project, was a 12-month effort with two major objectives.

- To determine the feasibility and value of reusable Ada software parts for missile flight software
- To determine the feasibility and value of automating (fully or partially) the process of building new missile flight software systems using parts

CAMP-1 started with a study to determine if sufficient commonality existed within missile flight software applications to warrant the development of reusable parts. After studying the operational flight software from ten existing missile systems, the CAMP team identified 250 common parts (during CAMP-2 this number grew to 454). Once these common parts were identified, their requirements were specified and their architectural designs were developed in accordance with DoD-STD-2167.

Concurrent with the identification, specification, and design of the reusable parts, the CAMP team performed an investigation to determine which aspects of building new software systems from parts could be automated. This investigation resulted in the definition and design of a tool known as a *parts composition system* (PCS) which would consist of three major subsystems.

- A Parts Identification subsystem which would help the user find parts applicable to his new application
- A Parts Catalog subsystem which would help the user understand and manage the available parts
- A Component Construction subsystem which consists of a set of tools to automatically generate reusable Ada code in situations where generated code was needed for reasons of efficiency, reusability, or ease of use. It also assists in the use of complex generic reusable Ada parts.

While CAMP-1 was primarily a feasibility study, CAMP-2 was primarily a technology demonstration. The main goal of the 30-month CAMP-2 project was to demonstrate the technical feasibility and value of reusable Ada missile parts and a PCS by building and using them on a realistic application.

The first major task in CAMP-2 was the construction of the reusable parts identified during CAMP-1. A total of 454 production-quality, reusable, Ada parts were coded, tested, and documented in accordance with DoD-STD-2167. The parts, together with their test code, consist of over forty thousand lines of Ada code. When completed, these parts were distributed to over 120 government agencies and contractors. Sections II, III, and VI of Volume I of this Final Technical Report discuss the construction of the CAMP parts in more detail.

A prototype of the parts composition system tool defined in CAMP-1 was also constructed, tested, and documented in accordance with DoD-STD-2167. To illustrate the utility of this tool, a user can spend 3 minutes describing his requirements for a Kalman filter subsystem and the tool will generate and assemble over 1900 lines of Ada code which efficiently implements this subsystem. Section IV of Volume I of this Final Technical report describes the construction of this prototype in more detail.

An important part of the CAMP-2 project was the construction of a real missile navigation and guidance system using the CAMP parts and the prototype PCS tool. This software, known as the *11th Missile Application*ⁱ, consisted of over 21,000 Ada statements of which 18%ⁱⁱ was obtained by reusing the CAMP parts. This software was cross-compiled using an Ada/1750A compiler and executed on 1750A processors within a missile simulation. The 11th Missile Demonstration served as a proving ground not only for the CAMP parts and the parts composition system tool, but also for Ada/1750A compiler technology. Volume II of this Final Technical Report describes the 11th Missile Demonstration in more detail.

Another CAMP-2 task was the development of a suite of benchmarks that could be used to measure the effectiveness of Ada compilers for armonicsⁱⁱⁱ applications. These benchmarks are standard Ada software units which test a compiler's ability to deal with realistic armonics situations. Volume III of this Final Technical Report describes the armonics benchmarks in more detail.

All of the CAMP products — the parts, the prototype PCS tool, and the Armonics Benchmarks — are available to U.S. government agencies and qualified government contractors.

Given the pathfinding nature of the CAMP program, it is not surprising that many lessons were learned concerning Ada, reuse, and the status of Ada compilers. Section VIII contains a detailed discussion of these conclusions.

The good news is that the Ada programming language was proven to be a good language for RTE applications and for achieving reuse within these applications. The entire 11th Missile Application was constructed using only 21 lines of assembly code, and the reuse of standard parts shows the potential for improving productivity by 15%. Use of the parts and the parts composition system showed the potential for even greater productivity gains (up to 28% when the PCS Kalman Filter Constructor was used in addition to the parts).

The bad news is that many current generation Ada compilers still have problems correctly and efficiently handling the more advanced features of Ada. Of particular concern to the CAMP team were the problems surrounding the handling of Ada generic units (see Section VII). If not corrected, these problems with generic units could have serious detrimental impacts on reuse within DoD RTE applications. Two actions are needed to solve this problem. First, the Ada validation process must be amended to include more stringent tests concerning a compiler's ability to properly handle complex use of generic

ⁱThe CAMP team used 10 missiles to identify parts and saved an 11th missile to verify the parts, hence the terminology.

ⁱⁱThis number increases to 22% if the parts that were modified are also counted.

ⁱⁱⁱARMament electRONICS

units. Second, Ada compilers must include more powerful global optimization techniques. Until the problems are corrected, DoD mission-critical RTE Ada projects should establish a contractual relationship with their compiler developer in order to reduce risk to the project.

Tasking throughput is currently another potential problem area in Ada compiler code generation. Although there does not appear to be anything inherently inefficient in the Ada language requirements with respect to tasking, work on the 11th Missile Application revealed that care should be given to selecting the kinds of tasking facilities used in an application.

The CAMP program marks the first practical application of reusable Ada parts to DoD mission-critical RTE applications. The program demonstrated that, given mature Ada compilers, the benefits of software reuse — reduced software development cost and schedules and higher software quality — can be achieved without sacrificing efficiency. If these benefits can be achieved in the missile domain, they can be achieved in other RTE domains.

Table of Contents

Section	Title	Page
I	INTRODUCTION	1
	1. Purpose	1
	2. Background	1
	3. Overview of the CAMP-1 Project	2
	4. Overview of the CAMP-2 Project	3
	5. Organization of the Report	6
II	DEVELOPMENT AND TESTING OF CAMP PARTS	7
	1. Terms and Structure	7
	2. Parts Development Methods	9
	a. Design and Code	10
	b. Testing	14
	c. Maintenance	15
	d. Configuration Management	16
	e. Tools	17
	(1) Design and Code Development Tools	18
	(2) Testing Tools	18
	(3) Configuration Management Tools	21
	(4) Documentation Tools	21
	(5) Miscellaneous Tools/Aids	22
	f. Documentation	23
	3. CAMP Parts Process Analysis	25
III	INTER-RELATIONSHIPS BETWEEN CAMP PARTS	30
	1. Parts Build On Other Parts	30
	2. Parts Work Together	32
	3. CAMP Parts Facilitate Use of Other Parts	33
IV	DEVELOPMENT AND TESTING OF A PARTS COMPOSITION SYSTEM (PCS)	41
	1. PCS Functionality	41
	a. Parts Catalog	42
	(1) Design	43
	(2) Testing and Operational Evaluation	46
	b. Parts Identification	47
	(1) Application Approach	47
	(2) Architectural Approach	48
	(3) Testing and Operational Evaluation	51
	c. Component Constructors	52

Table of Contents (cont'd)

Section	Title	Page
	(1) Design Paradigms	54
	(2) Constructor Implementation	58
	(a) Types of Constructors	58
	(b) Code Generation	58
	(3) Testing and Operational Evaluation	59
2.	PCS Implementation	59
a.	System Architecture	59
(1)	Hardware	60
(2)	Software	61
(3)	User Interface	62
b.	Parts Catalog	63
c.	Parts Identification	64
d.	Component Constructors	64
3.	Future Directions	64
V	THE ADA LANGUAGE AND SOFTWARE REUSABILITY	67
1.	Separate Compilation and Generic Units	67
2.	Optimization	70
3.	Task Priorities	72
4.	Address Clauses	72
5.	Implementation of Reduced-Precision Floating Point Types	73
6.	Procedural Data Types	75
7.	Dynamic Binding of Bodies to Specs	75
8.	Separation of Representation Clauses	79
VI	PARTS DESIGN METHODOLOGY	82
1.	Design Requirements	82
2.	Design Methods	83
a.	Typeless Method	84
b.	Overloaded Method	85
c.	Generic Method	86
d.	Abstract State Machine Method	88
e.	Abstract Data Type Method	89
f.	Skeletal Code Method	91
3.	Use of the Generic Method	91
a.	Using the Generic Method to Design Parts	93
b.	Using Parts to Construct an Application	95

Table of Contents (cont'd)

Section	Title	Page
	4. Semi-Abstract Data Type	98
	5. Summary	100
VII	ADA COMPILER VALIDATION AND SOFTWARE REUSABILITY	101
	1. Introduction	101
	2. Discussion	101
	a. A Sample System	102
	b. CAMP Experience With Ada Compilers	106
	c. Compiler Validation	110
VIII	CONCLUSIONS AND RECOMMENDATIONS	111
	1. On The Appropriateness of Ada for Reusable Software	111
	2. On The Appropriateness of Ada for Real-Time Embedded Reusable Software	113
	a. On the Effectiveness of Ada	113
	b. On the Inherent Efficiency of Ada	115
	c. On the Effectiveness of Ada Compilers	115
	d. On the Efficiency of Ada Compilers	117
	3. On The Development of the CAMP parts	119
	4. On The Benefits of using Parts	122
	5. On The Cost-Effectiveness Of Capturing Schematic Commonality	122
	6. On the Cataloging of Parts	123

Table of Contents (CONCLUDED)

Appendix	Title	Page
A	PARTS DATA BASE	125
	1. Introduction and Background	125
	2. ORACLE Relations	125
	3. Data Base Issues	142
	4. Conclusions and Recommendations	142
B	CATALOG ATTRIBUTES	145
	References	157

List of Figures

Figure	Title	Page
1	A Generic TLCSC Can Be A Part	8
2	Generic LLCSCs and Functions Can Be Parts	8
3	A Nongeneric Unit Can Be A Part	9
4	For High-Level Parts, Detailed Design is Code	12
5	Simple Parts Require Few Comments	12
6	Complicated Parts Require More Comments	13
7	CAMP Parts Testing Cycle	15
8	Sample Expected Results File	20
9	Sample Code Counter Input	23
10	Top-Level Design Header Information	24
11	Detailed Design Header Information	25
12	Lines of Code versus Ada Statements	26
13	CAMP Parts Sizing Data	26
14	CAMP Parts Effort Data	27
15	CAMP Parts Productivity Data	28
16	CAMP Parts Development Statistics	28
17	Assembling a North-Pointing Navigation System	31
18	Some Parts Build On Other Parts	32
19	Parts Work Together	33
20	CAMP Parts Facilitate Use of Other Parts	34
21	Required Operations Obtained Through Use of Generic Formal Parameters	37
22	Sample Instantiations of Geometric_Operation Parts Using Default Routines	38
23	Sample Instantiations of Geometric_Operation Parts Using Specialized Sin_Cos Procedure	39
24	Catalog Attributes	42
25	Parts Catalog Functions	43
26	Required Catalog Attributes	44
27	Searchable Catalog Attributes	45
28	Application Exploration	48
29	Application Exploration Example	50
30	Missile Model Walkthrough	50
31	Constructor Design Paradigm	55
32	Screen Flow Symbology	56
33	Kalman Filter Constructor — High-Level View	57

List of Figures (cont'd)

Figure	Title	Page
34	AMPEE System Architecture	60
35	Ada Generic Compilations	69
36	Methods of Generic Instantiation	71
37	Subtypes Should Support Reduced-Precision Operations	74
38	Partial Standard_Trig Package Specification	76
39	Partial Polynomials Package Specification	76
40	System Functions Version of Standard_Trig Package Body	77
41	Single Precision Version of Standard_Trig Package Body	78
42	Extended Precision Version of Standard_Trig Package Body	78
43	Multiple Precision Version of Standard_Trig Package Specification	79
44	11th Missile Application Use of Representation Clauses	80
45	Reusable Parts Methods	83
46	Strong Data Typing Example	85
47	Typeless Method	85
48	Overloaded Method	85
49	Generic Method	87
50	Tunneling of Parameters	87
51	State Machine Method	88
52	Abstract Data Type Method	90
53	Skeletal Code Template Method	91
54	Comparison of the Six Reusable Parts Methods	92
55	Commonality Captured in the Generic Part Body	94
56	Mechanism for Overriding Defaults	95
57	Autopilot Part Generic Specification	96
58	Selections from CAMP Parts for Instantiation	97
59	Autopilot Bundle Structure	98
60	Kalman Filter Bundle Structure	100
61	Generic Units Can Be Very Simple	102
62	Some Generic Units Can Be Very Complex	103
63	Nested Generic Units Can Be Very Complex	104
64	Most Generic Units Have Minimal Complexity	105
65	Assembling a North-Pointing Navigation System	105
66	Some Compilers Couldn't Handle Type Derivations	107
67	Overloaded Operator Caused Problems for Compiler	108
68	Compilers Had Problems Finding Default Subprograms	109

List of Figures (cont'd)

Figure	Title	Page
B-1	Catalog Attributes	146
B-2	CAMP Parts Taxonomy	154

List of Tables

Table	Title	Page
1	CAMP Parts Taxonomy	10
2	Design Steps	14
3	Items Under Configuration Management	17
4	Software Development File Contents	24
5	Application Exploration — Required User Inputs	49
6	BIM_Error_Messages Contents and Storage Representation	81
A-1	Columns in the TLCSC Relation	126
A-2	Columns in the Adalevel Relation	126
A-3	CAMP Parts Sizing List	128
B-1	'Used to Build' and 'Built From' Attribute Relationships	155

List of Acronyms

ACS	Ada Compilation System
ACVC	Ada Compiler Validation Capability
AdaJUG	Ada/Jovial Users Group
ADL	Ada Design Language
AFATL	Air Force Armament Laboratory
AFB	Air Force Base
AI	Artificial Intelligence
AJPO	Ada Joint Program Office
AMPEE	Ada Missile Parts Engineering Expert (System)
AMRAAM	Advanced Medium Range Air-to-Air Missile
ANSI	American National Standards Institute
APSE	Ada Programming Support Environment
Armonics	Armament Electronics
ART	Automated Reasoning Tool
ASCII	American Standard Code for Information Interchange
BC	Bus Controller
BDT	Basic Data Types
BIM	Bus Interface Module
CAD/CAM	Computer-Aid Design/Computer-Aided Manufacturing
CAMP	Common Ada Missile Packages
CCCB	Configuration Change Control Board
CDRL	Contractual Data Requirements List
CMS	Code Management System
ConvFactors	Conversion_Factors (TLCSC)
CPDS	Computer Program Development Specification
CPPS	Computer Program Product Specification
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CVMA	Coordinate_Vector_Matrix_Algebra (TLCSC)
DACS	Defense Analysis Center for Software
DBMS	Data Base Management System
DCL	DIGITAL Command Language
DDD	Detailed Design Document
DEC	Digital Equipment Corporation
DMA	Direct Memory Access
DoD	Department of Defense

DoD-STD	Department of Defense Standard
DPSS	Digital Processing Subsystem
DSR	Digital Standard Runoff
DTM	DEC /Test Manager
FMS	Forms Management System
FORTRAN	FORMula TRANslation
GPMath	General_Purpose_Math (TLCSC)
HOL	Higher-Order Language
Hr	Hour
I/O	Input/Output
ISA	Inertial Sensor Assembly
JOVIAL	Jules Own Version of International Algebraic Language
LISP	List Processing (language)
LLCSC	Lower Level Computer Software Component
LOC	Lines of Code
MDAC	McDonnell Douglas Astronautics Company
MDAC-HB	McDonnell Douglas Astronautics Company - Huntington Beach
MDAC-STL	McDonnell Douglas Astronautics Company - St. Louis
MDC	McDonnell Douglas Corporation
MIL-STD	Military Standard
MRASM	Medium Range Air-to-Surface Missile
NM	Nautical Miles
NPNav	North_Pointing_Navigation_Parts (TLCSC)
OCU	Operator Control Unit
Opns	Operations
PC	Personal Computer
PCS	Parts Composition System
PDL	Program Design Language
R&D	Research and Development
RT	Remote Terminal
RTE	Real-Time Embedded
SDF	Software Development File
SDI	Strategic Defense Initiative
SDN	Software Development Notebook
SDR	Software Discrepancy Report
SEAFAC	System Engineering Avionics Facility
SEI	Software Engineering Institute

SEP/SCP	Software Enhancement Proposal/Software Change Proposal
SIGAda	Special Interest Group on Ada
SRS	Software Requirements Specification
STARS	Software Technology for Adaptable, Reliable Systems
stmt	statement
SURMOS	Start-Up Real-time Multi-tasking Operating System
TLCSC	Top-Level Computer Software Component
TLDD	Top-Level Design Document
UnivConst	Universal_Constants (TLCSC)
VAX	Virtual Address Extension
VMS	Virtual Memory System
WGS72	World Geodetic System, 1972

SECTION I

INTRODUCTION

1. PURPOSE

This report contains a description of the work performed, the results achieved, and the lessons learned on the Common Ada Missile Packages Phase 2 (CAMP-2) project. CAMP was a multi-year research effort in which the McDonnell Douglas Astronautics Company-St. Louis (MDAC-STL) demonstrated the feasibility and value of reusable Ada software parts in embedded, real-time, mission-critical, DoD applications. This was accomplished by (a) building a library of efficient and reusable Ada parts for missile flight applications, (b) building a prototype parts composition system (PCS), and (c) testing the parts and the PCS by using them on an actual missile application.

The CAMP project has been sponsored by the Air Force Armament Laboratory at Eglin Air Force Base, and partially funded by the Air Force Armament Division; the DoD Software Technology for Adaptable, Reliable Systems (STARS) Program Office; and the Air Force Electronic Systems Division. The Ada Joint Program Office (AJPO) sponsored the initial distribution of CAMP software to 120 Government agencies and contractors. This software is now available through the Air Force Defense Analysis Center for Software (DACS) at Griffiss Air Force Base, New York.

2. BACKGROUND

Reusable software is rapidly becoming a key element in the plans of many Department of Defense (DoD) organizations to bring about a new software engineering environment that will result in higher quality software at a lower cost. The recently formed Software Engineering Institute (SEI) believes that *"a significant portion of the transition of new software engineering technology, the goal of the SEI, will be embodied in reusability and automation concepts"* (Reference 1). In a similar vein, the DoD Software Technology for Adaptable, Reliable Systems (STARS) program intends to *"develop a significant foundation of reusable Ada software ... for ... applications and software engineering support"* (Reference 2). Software reuse has even been identified as a major management issue by a DoD directive (Reference 3) on the management of computer resources in defense systems.

While many factors have influenced the recent wide-spread adoption of reusable software within the DoD, the most important factor has certainly been the Ada mandate. In 1983, the DoD mandated (Reference 4) the use of Ada as the standard programming language for mission-critical computer systems. This mandate was recently formalized in a pair of DoD directives (References 5 and 6).

Many software engineers who in the past have doubted the practicality of software reusability saw that, with a standard language such as Ada, meaningful levels of software reuse were within reach for the first time. However, not everyone within the DoD community believes that software reusability is feasible. One very important group that is not convinced of the practicality of reusability is the real-time embedded (RTE) software engineering community.

It has been a long-held tenet of the RTE community that software parts (i.e., components specifically

written to be reused) are not practical in real-time embedded applications. This community believes that software parts must be general to be reusable and that generality implies inefficiencies. While the non-RTE software engineer is usually willing to sacrifice some run-time efficiency for significant increases in software quality and productivity, the RTE software engineer often cannot afford this luxury. A typical RTE software engineer develops software for micro-computers embedded in products such as aircraft, missiles, and satellites. He cannot freely add more memory or upgrade to a more powerful processor since his computer must comply with severe limitations on weight, power requirements, and volume. Even recent advances in memory and processor technologies have not been of much help to the RTE software engineer since the demand for more functionality in his products more than account for the added capabilities provided by these technologies.

In order to convince the RTE software engineering community that software parts can work efficiently in the real-time embedded domain, it is essential that objective data be developed showing that software reuse is feasible. It is not enough to show that reusability works well in non-RTE applications. Given the pervasiveness of RTE computer applications within DoD mission-critical systems, it is imperative that questions about the feasibility of software parts be addressed squarely within the RTE domain, and preferably by means of a realistic demonstration. This was precisely the goal of the CAMP-2 project.

3. OVERVIEW OF THE CAMP-1 PROJECT

The CAMP program was initiated in 1984 with the award of the CAMP-1 project to MDAC-STL. The CAMP-1 project was a 12-month feasibility study with two major objectives: (a) to determine the feasibility and value of reusable Ada software parts for missile flight software, and (b) to determine the feasibility and value of automating (fully or partially) the process of building new missile flight software systems using parts.

The CAMP-1 Final Technical Report contains a detailed description of the tasks performed and results obtained during that project. It can be obtained from the Defense Technical Information Center using the following access numbers: AD-B-102 654 (Volume 1), AD-B-102 655 (Volume 2), and AD-B-102 656 (Volume 3). The major tasks performed during CAMP-1 were as follow:

- Domain Commonality Analysis: The purpose of this analysis was to determine if sufficient commonality existed to justify the development of reusable Ada missile parts. Ten missiles were studied with the result being the identification of over 200 reusable Ada software parts. A Software Requirements Specification (SRS) was prepared for these parts in accordance with DOD-STD-2167.
- Ada Part Design: After the parts were identified, their architectural designs were developed and documented in a Software Top-Level Design Document (STLDD) in accordance with DOD-STD-2167.
- Part Composition System (PCS) Investigation: The purpose of this investigation was to determine which aspects of building new software systems from parts could be automated. The result of this investigation was the development of an SRS for a prototype tool called the Ada Missile Parts

Engineering Expert (AMPEE) System. The goal of this tool was to help the software engineer find, understand, use, and manage the reusable Ada missile parts.

- AMPEE Design: After the requirements were specified, the architectural design of the AMPEE system was developed and documented in a STLDD.

4. OVERVIEW OF THE CAMP-2 PROJECT

While CAMP-1 concentrated on feasibility analyses, CAMP-2 was primarily a technology demonstration. CAMP-2 was a 30-month project which began in September, 1985. The overall goal of CAMP-2 was to demonstrate the technical feasibility and value of reusable Ada missile parts and a PCS by building and using them on a realistic application. The following tasks were performed on CAMP-2:

- Parts Construction: The purpose of this task was to develop the detailed design of the parts which were identified during CAMP-1, and to code and test the parts. It was during this task that additional parts were identified, bringing the total number of parts developed to 454.
- AMPEE Construction: During this task, the detailed design of the prototype parts composition system was developed, and the system was coded and tested.
- 11th Missile Application Development: This task involved the construction of an actual missile application using the Ada parts and the AMPEE system, and testing of the developed system in a 1750A hardware-in-the-loop simulation.
- Armonics Benchmarks: The purpose of this task was to use the CAMP parts to develop a suite of benchmarks that could be used to measure the effectiveness and efficiency of Ada compilers for armonics¹ applications.

The CAMP-2 products included deliverable software, software documentation, and new software technology. CAMP software may be obtained by certified government contractors and government agencies by writing to the Air Force Rome Air Development Center/Data and Analysis Center, (315) 336-0937. CAMP documents listed below with Air Force Armament Laboratory Technical Report numbers may be ordered from the Defense Technical Information Center.

¹ARMament electRONICS

1. PARTS PRODUCTS: Over 450 efficient, reusable Ada parts for missile flight applications.

- a. User's Guide: A listing of all parts, their purpose and decomposition, other parts required for their use, where they may be used in other instantiations, etc. (AFATL-TR-88-18, Volume 1)
- b. Version Description Document: A document containing an inventory of distribution items, installation instructions, and other information. (AFATL-TR-88-18, Volume 2)
- c. Software Product Specification: As-built versions of all specifications in accordance with DOD-STD-2167. (AFATL-TR-88-18, Volume 3)
- d. Top-Level Design Document: The architectural design (updated from CAMP-1) for the CAMP parts documented in accordance with DOD-STD-2167. (AFATL-TR-88-18, Volumes 4-6)
- e. Detailed Design Document: The detailed design for the CAMP parts documented in accordance with DOD-STD-2167. (AFATL-TR-88-18, Volumes 7-12)
- f. Test Plan: The plan by which the parts were tested in accordance with DOD-STD-2167. (AFATL-TR-88-22)
- g. Test Procedure: The procedures by which the parts were tested in accordance with DOD-STD-2167. This was tailored to include information that would usually be found in the Software Test Description and the Software Test Report. (AFATL-TR-88-23, Volumes 1-8)
- h. Software Development Files: The working development notebooks containing source code listings, test plan, test procedure, test code, and test results for the CAMP parts in accordance with DOD-STD-2167.
- i. Parts Tape: An ANSI standard tape containing source code for the parts, test code and utilities, and design documents in machine readable form.
- j. Parts Sizing List: A microfiche containing sizing data about all parts.

2. AMPEE SYSTEM PRODUCTS: A prototype software parts composition tool including a parts catalog, a parts identification facility, and a component construction facility.

- a. Software Product Specification: As-built versions of all specifications documented in accordance with DOD-STD-2167. This included source code listings for the AMPEE system. (AFATL-TR-88-19, Volume 1)
- b. Top-Level Design Document: The architectural design (updated from CAMP-1) for the AMPEE system documented in accordance with DOD-STD-2167. (AFATL-TR-88-19, Volume 2)

- c. Detailed Design Document: The detailed design for the AMPEE system documented in accordance with DOD-STD-2167. (AFATL-TR-88-19, Volume 3)
- d. Parts Catalog: Printed form of all data stored in the AMPEE system catalog. (AFATL-TR-88-20, Volumes 1-4)
- e. User's Manual: A manual providing the user with detailed instructions on the use of the AMPEE system. (AFATL-TR-88-21)
- f. Test Plan: The plan by which the AMPEE system was tested in accordance with DOD-STD-2167. (AFATL-TR-88-22)
- g. AMPEE Tape: A tape containing source code for the AMPEE system, utilities, and the catalog files.
- h. Training Plan: A plan which was used to develop training in the use and maintenance of the AMPEE system.

3. 11TH MISSILE DEMONSTRATION PRODUCTS: A complete missile navigation and guidance application built using CAMP parts and the AMPEE system, and tested in a 1750A hardware-in-the-loop simulation.

- a. Software Requirements Specification: The requirements of the missile application documented in accordance with DOD-STD-2167. (AFATL-TR-88-24, Volume 1)
- b. Top-Level Design Document: The architectural design for the 11th Missile system documented in accordance with DOD-STD-2167. (AFATL-TR-88-24, Volume 2)
- c. Test Plan: The plan by which the 11th Missile system was tested in accordance with DOD-STD-2167.
- d. Test Report: The results of testing the application in accordance with DOD-STD-2167. This includes 11th Missile development evaluation.

4. ARMONICS BENCHMARK PRODUCTS: A self-documenting set of tests to be run for evaluation of Ada development and run-time environments within armonics applications

- a. Benchmark Tape: An ANSI tape containing the benchmarks, standard data files, and VAX command procedures for executing the benchmarks on VAX hardware.

5. OTHER PRODUCTS

- a. Final Technical Report: Three volumes covering parts and PCS development, 11th Missile Application development, and Armonics Benchmarks development
- b. Monthly Status Reports and Schedule: Management reports
- c. Program Status Reviews: Slides used at periodic status reviews

- d. SIGAda Demonstration: Slides used at a series of one-hour presentations of CAMP technology
- e. AFATL Demonstration: Slides used at a series of three-hour presentations of CAMP technology

5. ORGANIZATION OF THE REPORT

Due to the large amount of data to be discussed in this report, it has been divided into three volumes. The remaining sections of Volume I are organized as follows.

- Section II describes the development and testing of the CAMP parts
- Section III goes into additional detail regarding the inter-relationships between some of the CAMP parts
- Section IV describes the development and testing of the AMPEE system
- Section V discusses some issues concerning the Ada language and their impact on reusable software
- Section VI describes the methodology used in designing the CAMP Ada parts
- Section VII describes a problem with current Ada compilers that potentially could have a major adverse effect on reusable software
- Section VIII contains overall conclusions and recommendations

Volume II describes the development and testing of the 11th Missile Application. Volume III describes the development and testing of the Armonics Benchmarks.

SECTION II

DEVELOPMENT AND TESTING OF CAMP PARTS

Prior to the CAMP program, there were no successful projects to carry the development of a general library of reusable, real-time embedded software through the software lifecycle. In fact, except for tool catalogs and abstract data types, no complete software library existed in Ada. Therefore, during the early stages of the CAMP parts development, many new issues connected with the development of reusable software had to be addressed.

These issues included: 1) definition of terms, 2) the basic structure to be used when designing the parts, and 3) documentation standards for the parts. The CAMP team had to define a common terminology because discussing the number of parts that had been developed or how parts had been packaged in TLCSCs and LLCSCs has little meaning without a common understanding of what constitutes a part, a TLCSC, and an LLCSC. Development of the parts could not proceed until the basic design approach and structure of the parts had been decided. Finally, due to the large number of parts, it was necessary to determine how to satisfy documentation requirements within practical limits.

I. TERMS AND STRUCTURE

One issue that was addressed during the CAMP project is what actually constitutes a part: is it a package, is it an executable unit, is it a compilation unit, etc. Various definitions of a part had been given in the past; for example, parts had been defined as Ada units (e.g., packages, procedures, functions), design units, and code units, with or without test code. While these were not incorrect definitions, they were not appropriate for CAMP. The criteria established on CAMP for determining if a piece of code was a part are enumerated below. Using these criteria, 454 Ada parts were developed during the CAMP program.

1. A part is a package, subprogram, or task. A part can be a Top-Level Computer Software Component (TLCSC), Lower Level Computer Software Component (LLCSC), or unit. A TLCSC is defined as an outer level package or procedure — one that was not nested in another package. An LLCSC is defined as a package that is nested in some other entity, generally within another package. Units are defined as nested procedures, functions, or tasks.
2. A part must be usable in a stand-alone fashion.
 - It may *with* other parts.
 - It does not depend on other packages, subprograms, or tasks encapsulated with it to perform a single function.

Figure 1 shows an example of a generic TLCSC, Clock_Handler, that is a part. The Clock_Handler package maintains a clock. Even though a single application may not require all of the routines in Clock_Handler, the routines could not logically exist alone: it would make little sense to reset a clock that is never read. Therefore, the entire TLCSC is considered a part.

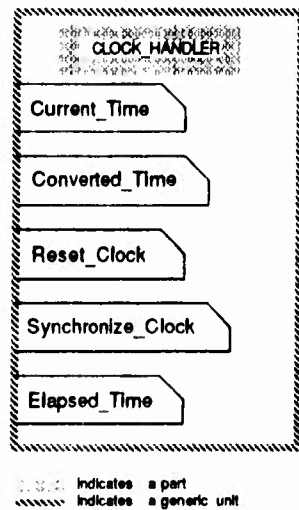


Figure 1. A Generic TLCSC Can Be A Part

Figure 2 shows an example of a generic LLCSC, Latitude_Integration, that is a part. This package maintains a latitude. The LLCSC is designated as a part because, although the Integrate function could exist on its own, the Reinitialize procedure could not.

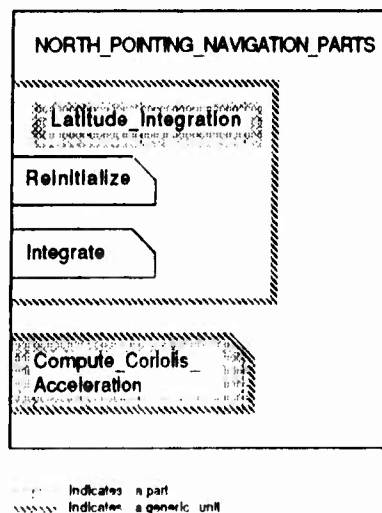


Figure 2. Generic LLCSCs and Functions Can Be Parts

Figure 2 also shows an example of a generic procedure, Compute_Coriolis_Acceleration, which is a part. Figure 3 shows an example of a generic package, Vector_Operations,

which contains several subroutines, each of which is a part. In these cases, the procedures, rather than any encapsulating packages, are designated as parts since the procedures can logically exist on their own.

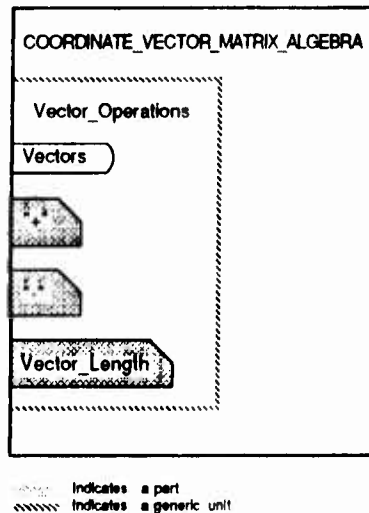


Figure 3. A Nongeneric Unit Can Be A Part

- A part may require types or objects that have been encapsulated with it: The subroutines shown in Figure 3 are parts even though they require the data type, **Vectors**, defined by the **Vector_Operations** package.

3. Organizational packages are not parts; and package bodies are never parts, even if they have processing within them

Given the large number of parts typically identified during any domain analysis, it is useful to develop some type of software parts taxonomy. This taxonomy provides a means of classifying parts; it helps not only domain analysts, but also helps users identify available parts. Table 1 lists the categories in the CAMP parts taxonomy, and includes a description of the classes and a listing of the TLCSCs belonging to each class.

2. PARTS DEVELOPMENT METHODS

CAMP-1 included a domain analysis to identify commonality between ten missiles which were studied. Following the domain analysis, requirements were defined for the common functions that were identified, and parts development began. Development was completed during CAMP-2. The development cycle included top-level design, detailed design, coding, testing, and documentation. These development activities are further discussed in the following paragraphs.

TABLE 1. CAMP PARTS TAXONOMY

CATEGORY	TLCSC NAME	DESCRIPTION
Data Constants	WGS72_Ellipsoid_Engineering_Data WGS72_Ellipsoid_Metric_Data WGS72_Ellipsoid_Unitless_Data Universal_Constants Conversion_Factors	TLCSCs which provide data constants used in a typical missile application
Data Types	Basic_Data_Types Kalman_Filter_Data_Types Autopilot_Data_Types	TLCSCs which provide data types used in other TLCSCs or in a user application
Equipment Interfaces	Missile_Radar_Altimeter Missile_Radar_Altimeter_with_Autopower_On Clock_Handler	TLCSCs which provide standard interfaces to specific hardware components or to general classes of hardware
Navigation	Common_Navigation_Parts Wander_Azimuth_Navigation_Parts North_Pointing_Navigation_Parts Direction_Cosine_Matrix_Operations	TLCSCs which provide the basic functionality of a navigation subsystem
Kalman Filter	Kalman_Filter_Common_Parts TLCSC Kalman_Filter_Compact_H_Parts TLCSC Kalman_Filter_Complicated_H_Parts TLCSC	TLCSCs which provide common Kalman filter functions
Guidance and Control	Waypoint_Steering Autopilot	TLCSCs which provide the basic functionality of a guidance and control subsystem
Nonguidance Control	Air_Data_Parts TLCSC Fuel_Control_Parts TLCSC	TLCSCs which provide the basic functionality of a control subsystem for operations outside of the guidance area
Mathematical	Coordinate_Vector_Matrix_Algebra General_Vector_Matrix_Algebra Standard_Trig Geometric_Operations Signal_Processing Polynomials General_Purpose_Math Unit_Conversions External_Form_Conversion_Two's_Complement Quaternion_Operations	TLCSCs which provide a variety of useful mathematical functions such as coordinate and matrix algebra, trigonometric, and signal processing functions
Abstract Mechanisms	Abstract_Data_Structures	TLCSCs which provide abstract data structures and processes
General Utilities	General_Uilities Communication_Parts	TLCSCs which provide other functions needed for missile or other weapons system operation

a. Design and Code

On CAMP, top-level design consisted of the package specifications for all the CAMP parts TLCSCs, including the specifications for all exported LLCSCs and units, as well as the definition of all exported data types, constants, and exceptions.

Detailed design and coding phases were merged through the use of Ada as the design language. The primary purpose of the program design language (PDL), Ada design language (ADL), and/or pseudocode developed during detailed design is to improve understanding of the software by providing additional information that is an appropriate level of abstraction above the code. The key here is that detailed design should be a higher level of abstraction than the code. If it is not, then there may be excessive duplication of effort during the detailed design and coding phases. There were certain charac-

teristics of the CAMP project which led to the conclusion that it was appropriate to go directly from top-level design to code for development of the CAMP parts. These characteristics are discussed below.

- Low-level requirements: The requirements for many of the parts were specified at a very low level. The algorithms to be used in many of the math parts, for example, were completely specified during the requirements phase. There was, therefore, no need to repeat these algorithmic requirements in the detailed design.
- Parts were built of other parts: Many of the high-level CAMP parts were designed to accept other parts as generic parameters. The highest level parts directly instantiate the CAMP parts required to perform lower level operations. These design aspects of the CAMP parts are further discussed in Sections III and VI.

An example of parts instantiating other parts is shown in Figure 4 which contains the detailed design/code for the Kalman_Filter_Complicated_H_Parts.Sequentially_Updated_Covariance_Matrix_and_State_Vector.Update procedure. The English pseudocode for this procedure would be similar to the following:

```
for each measurement in the state vector loop
  compute K (Kalman gain)
  update P (error covariance matrix)
  update X (state vector)
end loop
```

The actual code for this procedure would be very similar to the pseudocode if, as in the CAMP parts, the calculations of a new K, P, and X consisted simply of calls to other routines. This similarity can be seen by comparing the above pseudocode with the actual code, shown in Figure 4, which contains nothing more than a loop and three subroutine calls.

- Parts are small: The CAMP parts tend to be small, less than 36 lines of code on average, and, therefore, relatively simple. The need for high-level comments frequently decreases as the simplicity of the code increases. This can be seen in Figures 5 and 6 which contain simple and relatively complex routines, respectively. Figure 5 shows a piece of code which sets all elements of a symmetric, full-storage matrix to 0.0. The code for this procedure is quite simple and self-explanatory; it, therefore, contains no comments other than those in its code header. Figure 6, on the other hand, contains a more complicated piece of code which subtracts a symmetric, full-storage matrix from an identity matrix. Because of the complexity of this code, high-level comments, in addition to those contained in its code header, were required. This ratio of comments to code for this piece of code is better than 1:2.

While this merging of the detailed design and coding phases, hereafter referred to as detailed design, is not appropriate for all applications, it was appropriate for development of the CAMP parts.

The primary steps during the design phases are shown in Table 2.

Design walkthroughs were attended by all members of the CAMP parts team and occasionally

```

package body Kalman_Filter_Complicated_H_Parts is

function Compute_Kalman_Gain ...
procedure Update_Error_Covariance_Matrix ...
procedure Update_State_Vector ...

package body Sequentially_Update_Covariance_Matrix_and_State_Vector is

K : K_Column_Vectors;

function Compute_K is new Compute_Kalman_Gain ...
procedure Update_P is new Update_Error_Covariance_Matrix ...
procedure Update_X is new Update_State_Vector ...

procedure Update (P                : in out P_Matrices;
                  X                : in out State_Vectors;
                  Z                : in      Measurement_Vectors;
                  Complicated_H    : in      H_Matrices;
                  Measurement_Variance : in      Measurement_Variance_Vector) is

begin

  for Measurement_Number in Measurement_Indices loop

    K := Compute_K (P                => P,
                   Measurement_Number => Measurement_Number,
                   Complicated_H      => Complicated_H,
                   Measurement_Variance => Measurement_Variance);

    Update_P (P                => P,
             Measurement_Number => Measurement_Number,
             K                  => K,
             Complicated_H      => Complicated_H);

    Update_X (X                => X,
             Z                  => Z,
             K                  => K,
             Measurement_Number => Measurement_Number,
             Complicated_H      => Complicated_H);

  end loop;

end Update;

end Sequentially_Update_Covariance_Matrix_and_State_Vector;

end Kalman_Filter_Complicated_H_Parts;

```

Figure 4. For High-Level Parts, Detailed Design is Code

```

separate (General_Vector_Matrix_Algebra.
         Symmetric_Full_Storage_Matrix_Operations_Unconstrained)
procedure Set_To_Zero_Matrix (Matrix : out Matrices) is

begin

  Matrix := (others => (others => 0.0));

end Set_To_Zero_Matrix;

```

Figure 5. Simple Parts Require Few Comments

```

function Subtract_from_Identity (Input : Matrices) return Matrices is
-- -----
-- --declaration section
-- -----

    Answer : Matrices(Input'RANGE(1), Input'RANGE(2));
    Col     : Col_Indices;
    Col_Count : POSITIVE;
    Row     : Row_Indices;
    Row_Count : POSITIVE;
    S_Col    : Col_Indices;
    S_Row    : Row_Indices;

-- -----
--begin function Subtract_from_Identity
-- -----

begin
-- --make sure input matrix is a square matrix
    if Input'LENGTH(1) = Input'LENGTH(2) then

-- --will subtract input matrix from an identity matrix by first
-- --subtracting all elements from 0.0 and then adding 1.0 to the
-- --diagonal elements;
-- --when doing the subtraction, will only calculate the remainder
-- --for the elements in the bottom half of the matrix and will simply
-- --do assignments for the symmetric elements in the top half of the
-- --matrix

        Row_Count := 1;

-- --S_Col will go across the columns as Row goes down the rows;
-- --Will mark column containing the diagonal element for this row
        Row := Input'FIRST(1);
        S_Col := Input'FIRST(2);
        Do_Every_Row:
        loop
            Col_Count := 1;

-- --S_Row will go down the rows as Col goes across the columns;
-- --when paired with S_Col will mark the symmetric counterpart
-- --to the element being referenced in the bottom half of the
-- --matrix
            Col := Input'FIRST(2);
            S_Row := Input'FIRST(1);
            Subtract_Elements_From_Zero:
            loop
-- --perform subtraction on element in bottom half of matrix
                Answer(Row,Col) := - Input(Row,Col);

-- --exit loop after diagonal element has been reached
                exit Subtract_Elements_From_Zero when Col_Count =
                    Row_Count;

-- --assign values to symmetric elements in top half of matrix
-- --(done after check for diagonal, since diagonal elements
-- -- don't have a symmetric counterpart)
                Answer(S_Row,S_Col) := Answer(Row,Col);

-- --increment variables
                Col_Count := Col_Count + 1;
                Col := Col_Indices'SUCC(Col);
                S_Row := Row_Indices'SUCC(S_Row);
            end loop Subtract_Elements_From_Zero;

-- --add one to the diagonal element
                Answer(Row, Col) := Answer(Row, S_Col) + 1.0;
                Row_Count := Row_Count + 1;
                Row := Row_Indices'SUCC(Row);
                S_Col := Col_Indices'SUCC(S_Col);
            end loop Do_Every_Row;
        else
            raise Dimension_Error;
        end if;

        return Answer;
    end Subtract_from_Identity;

```

Figure 6. Complicated Parts Require More Comments

TABLE 2. DESIGN STEPS

STEP	DESCRIPTION
1.	Assignment of requirements to the TLCSC
2.	Completion of top-level design, along with header information (see Section II.2.f)
3.	Preparation of a software development file (SDF) (see Section II.2.f)
4.	Top-level design walkthrough
5.	Completion of detailed design, along with header information (see Section II.2.f)
6.	Preparation of test procedure/plan (see Section II.2.b)
7.	Detailed design walkthrough

by members of the parts composition team. The design presented for walkthrough was reviewed to ensure conformance with requirements, conformance of design with existing design and coding standards, consistency with other parts, completeness of documentation, and conformance of code headers to document generation tool requirements (see Section II.2.e.(4)).

b. Testing

The testing phase of the life cycle began after completion of detailed design and prior to the detailed design walkthrough. During this phase, a test plan and procedure for the TLCSC were prepared for later review by the CAMP parts team at the detailed design walkthrough. Following completion of all design walkthroughs and implementation of walkthrough action items, a part was given to a tester for unit/integration testing.

Unit and integration testing of the CAMP parts were combined into a single phase because of the bottom-up approach taken to testing. All parts requiring other parts directly or designed to use them through generic parameters were actually tested using the supporting parts which had already passed testing. This approach shortened the testing phase by eliminating the need to write code stubs and by eliminating the need to first test a part in isolation and then retest it using the parts themselves.

Most parts required several iterations through the testing cycle illustrated in Figure 7. The majority of testing errors resulted from errors in the test procedures. Much less frequently, errors were found in the parts. On rare occasions, errors were found in the supporting parts which had already been tested. If an error were found in a part, whether directly or indirectly, it was returned to the original designer for modifications and sent through the testing cycle again. When a TLCSC successfully passed testing, it was placed under configuration control (see Section II.2.d) and compiled into the main CAMP parts library.

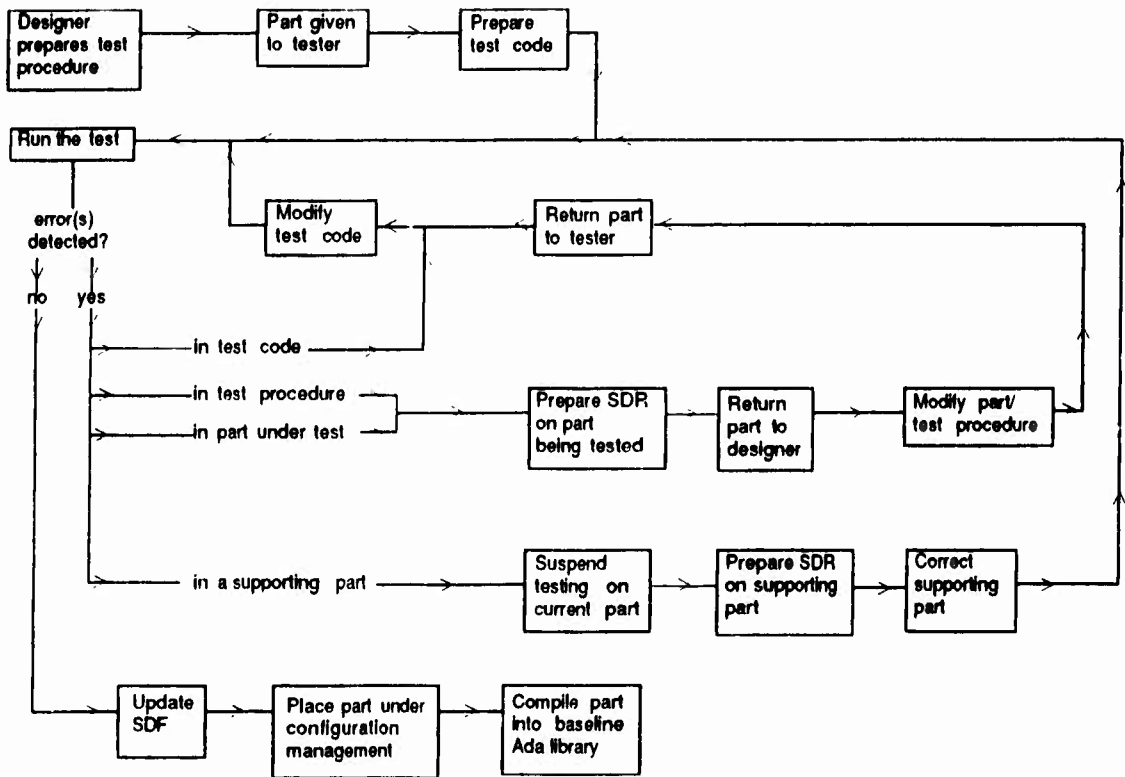


Figure 7. CAMP Parts Testing Cycle

c. Maintenance

During the CAMP project, parts were modified to provide both enhancements and corrections. Changes to the CAMP parts were governed by a Configuration Change Control Board (CCCB) that was put into effect after parts development was complete. The CCCB consisted of the program manager and the heads of the 11th Missile and parts development teams. On occasion, members of the parts composition system team and additional members of the parts development teams participated in board discussions. The CCCB was tasked to determine whether a proposed modification/enhancement to a part should be made. The outcome of the decision was based on:

- The scope of the change: Was it a minor change or a major one? Was it specific to the 11th Missile or general enough to be relevant to other missile systems?
- Purpose of the change: Was it to correct an error (errors were always corrected) or provide an enhancement?
- Schedule constraints

The need for corrections to the CAMP parts was determined at several points during the life cycle of the parts. Corrections due to errors detected during unit and integration testing are discussed in Section II.2.b. Occasionally, errors were detected in parts that had been successfully unit and integration tested. These errors were generally due to incorrect requirements and were identified through the 11th

Missile Application use of the CAMP parts, as well as through reviews of the parts by other McDonnell Douglas software projects for potential use in their systems. These errors were corrected as they were discovered. The affected parts were then retested and baselined again.

During the development of the 11th Missile Application, it was found that some parts, while not incorrect, were inappropriate for use on that project. Some of these inadequacies were due to requirements and some were due to design decisions. These problems were handled in one of the following ways:

- Baselined parts were modified: This course of action was chosen if it was determined that the parts were inappropriate not only for the 11th Missile Application, but also for other missile applications. For example, all the Kalman filter packages were modified because it was found that the generic parameters did not allow sufficient flexibility.
- Additional parts were created: The algorithms for some of the parts made assumptions that were not appropriate for the 11th Missile Application. For example, some of the navigation parts take advantage of the fact that for small angles, the sine of the angle is approximately equal to the angle itself. This assumption increases efficiency by eliminating the need to calculate an arcsine and produces satisfactory results for some missile applications. This assumption, however, was not appropriate for the 11th Missile Application and potentially not for other missiles either. Consequently, new parts were created which used the arcsine instead of the approximation.
- 11th Missile team modified their own versions of the parts: In some cases, the required modifications were specific to the 11th Missile Application and, therefore, did not warrant modifications to the baselined CAMP parts. In these instances, the 11th Missile team modified their own versions of the parts as required. A further discussion of this can be found in Volume II.

d. Configuration Management

Two libraries were created to aid in configuration management of all CAMP parts. These libraries were created under the DEC Ada Compilation System (ACS) and Configuration Management System (CMS). The ACS library contained compilations of the current versions of all baselined CAMP parts. The CMS library contained the ASCII files for all baselined CAMP parts. Both of these libraries were controlled by one member of the parts team: the parts librarian. Read access was given to all members of the CAMP team, but only the CAMP librarian could place elements in these libraries. The ACS and CMS tools are further discussed in Sections II.2.e.(1) and II.2.e.(3), respectively.

The CAMP librarian was responsible for baselining all CAMP TLCSCs. A TLCSC was baselined when it had successfully passed its testing phase, all source code documentation had been updated to include testing information, and the Software Development File (SDF) (see Section II.2.f) had been brought up to date.

When a TLCSC was first placed under configuration control, all files pertaining to the TLCSC were placed in the CMS library; these files included those listed in Table 3. The TLCSC was then compiled into the ACS library.

TABLE 3. ITEMS UNDER CONFIGURATION MANAGEMENT

CONTENTS	
1.	All source code files for TLCSC
2.	Test procedure
3.	Test plan
4.	All source code files containing test code
5.	Input data for tests
6.	Expected results for tests
7.	Results of testing
8.	DEC/Test Manager command files used to organize tests

If a TLCSC required modifications, the CAMP librarian would reserve the files requested by the person responsible for making the modifications. The files were checked back into CMS when the modifications were complete, the TLCSC was successfully retested, and the source code documentation and SDF were updated.

When rebaselining a modified TLCSC, the modified files were placed back into the CMS library; new files, if any, were placed under configuration control by placing them in the CMS library; the modified TLCSC was compiled into the ACS library; and any TLCSCs whose compilations depended upon the newly compiled TLCSC were recompiled.

c. Tools

Software tools were used by all members of the CAMP team during all phases of the project. This was a critical component in the increased productivity experience on the CAMP project. Some of the tools were provided by commercial vendors and satisfied standard needs such as library management, configuration management, symbolic debugging, editing, and text processing. In other areas, such as document production, requirements for tools were identified which could not be met with commercial products, and in-house tools were developed.

(1) Design and Code Development Tools

All CAMP Ada development took place using the Ada programming support environment (APSE) provided by Digital Equipment Corporation (DEC). This development environment includes: 1) the VAX Ada compiler; 2) the Ada Compilation System (ACS) which serves as the program library manager and provides an interface to the compiler and linker; and 3) a symbolic debugger.

The development environment provided by the Ada Compilation System facilitated the development of parts by multiple engineers. The ability to create sublibraries allowed the creation of one parent library containing all tested, baselined parts, and separate sublibraries for the untested software under control of the parts developers. The use of one parent and multiple sublibraries allowed all parts developers immediate access to baselined CAMP parts. It also gave the developers immediate access to all parts which were modified and, therefore, recompiled.

Unlike other development environments, ACS does not impose the restriction of requiring library unit specifications and bodies to be compiled into the same library. The compilation system also allows units to be entered via pointers from one library or sublibrary into another. This allows parts physically located in another library to be shared by reference. This method of entering rather than compiling a referenced unit into a library has the advantage of avoiding the problem of compiling against an obsolete version.

The usefulness of DEC's Ada Compilation System is enhanced by its integration with both the DEC Code Management System (CMS) and the symbolic debugger. This allows ACS to fetch files from the CMS library for recompilations. It also allows the symbolic debugger to fetch files from the Ada library in order to display source code lines during a debugging session. Both of these features were used extensively during CAMP.

Another useful and frequently used feature, is the ability of the ACS library manager to automatically perform recompilations of obsolete units. When invoking this feature, it is possible to indicate that a unit is to be considered obsolete if, in addition to the normal rules of compilation, the creation date of the latest source code file is more recent than the latest object code. This, along with integration of ACS with CMS, allows the library manager to retrieve files from CMS for recompilation whenever a new version of the part is baselined.

(2) Testing Tools

The testing phase of a part's life cycle included the identification and organization of required tests, preparation of a test plan and procedure, preparation of test code, and actual testing of the part. The tools which were used to assist with all of these phases are discussed below.

Test Manager

The VAX DEC/Test Manager (DTM) is a tool developed by DEC to assist in the organization of tests, selection of tests for execution, and review/verification of test results. It was used on CAMP to organize tests and assist in preparation of the test plan.

Tests were generally organized by creating a group of tests for each TLCSC and then creating subgroups for each LLCSC within a given TLCSC. The tests and groups were created by writing job control files containing the appropriate DTM commands and then submitting these files to the test manager. While DTM does have interactive capability, it was felt that the number and size of the required commands were too great for this capability to be practically applied, particularly considering the number of tests required for even a medium-sized TLCSC.

Following creation of the appropriate tests and groups for a TLCSC, DTM could be queried to show all the tests and groups for a particular case. A tool was written to take this output and create the tables which were used to document the tests for the test plan document.

An attempt was made to use the DEC/Test Manager for testing of the CAMP parts, but DTM proved unacceptable since it allowed no tolerance in the output. The results of a test had to be exactly what were expected or the test failed. For example, if the expected result was 2.0 and the actual result was 1.9999999999 or 2.00000000001, the test failed. Therefore, use of DTM for the execution of tests was discontinued.

Record Results and Retrieval Operations packages

During the early stages of CAMP parts testing, tools were developed to assist with the execution of tests. These tools consisted of the Record_Results and Retrieval_Operations packages.

The Record_Results package was designed to control the output file, retrieve data from the expected results file, format output to the results file, and check the results of each test. It consisted of several subroutines and several generic packages. The subroutines dealt with initializing the recording operations, opening and closing the output file, textual output to the file, formatting the file, and tailoring heading information. The generic packages were designed to handle floating point, integer, and enumeration data types and contained the actual recording/analysis routines.

The recording/analysis routines were overloaded to allow for variations in the recording operations themselves: whether the description was to be a textual description or simply a running count of the number of tests performed; whether the expected value was being sent to the routine or should be read from an expected results file. Each of the routines had a parameter controlling the tolerance to be used for judging every value recorded for each test. A value was considered acceptable if:

$$\text{abs (Actual - Expected)} \leq \text{abs(Expected)} * \text{Tolerance}$$

The recording routines were able to skip over extraneous text when retrieving data from the expected results file. Figure 8 shows an excerpt from the expected results file used for testing the Waypoint_Steering TLCSC. It was created by retaining applicable sections from the test procedure. The recording routines had the capability to go into a file such as the example, skip over the extraneous text, read the floating point values for UN_B, again skip over extraneous text, and read the enumeration values for the Start_Test function without having any knowledge of the textual format of the file. Being able to do this had several benefits:

- Testing was simplified: Since the expected results file was a trimmed-down version of the test procedure, complete with pertinent paragraph headings, it was easy to tell whether the numbers being read for a particular test were the ones that were supposed to be read.
- Time was saved: There were definite advantages to being able to have extra text in the expected results file, but it would have been inconvenient to have required a rigid format or to have had the test code know the textual layout of the file. By creating routines capable of skipping over superfluous data without knowing the format, time was saved. Additional time savings were also realized by creating a tool capable of assisting in the job of stripping the test procedure to create the expected results file.

```
x.2.2 FORMAL TEST X.X.X - UPDATE PROCEDURE

x.2.2.5 OUTPUT
Execution should generate the following output:
--first set of results
0.287_603_734_197 0.197_187_132_186 -0.937_536_950_976 --UN_B values

x.9.2 FORMAL TEST X.X.X - START_TEST FUNCTION

x.9.2.5 OUTPUT
Execution should generate the following output:
Not_Turning
Turning
```

Figure 8. Sample Expected Results File

Text Formatter

Digital Standard Runoff (DSR) is a text formatting tool supplied by the Digital Equipment Corporation. It processes source files into formatted text, optionally creating a table of contents. DSR was used on CAMP for the creation of test procedures, the top-level design document, and the detailed design document.

Symbolic Debugger

The DEC Ada Compilation System includes a symbolic debugger. The functions of the VAX symbolic debugger include the ability to run programs, set breakpoints, and execute individual instructions; examine, set, and evaluate program data; and show a trace of active calls at the current program counter location. It permits debugging in a screen mode which placed source code in one

window and debugger commands and output in another. Since the debugger recognizes Ada constructs, it was possible to ask for the current value of a component of an array or record, or to ask it to evaluate the attribute for some object or type.

The symbolic debugger did have a few limitations:

- Variable initialization: The symbolic debugger apparently initializes some variables when it is invoked; this causes difficulties in locating program errors. For example, one program was abnormally terminating due to a constraint error. When an attempt was made to identify the problem using the symbolic debugger, the program ran successfully. It took several iterations of running/debugging before it was realized that the program ran successfully in the debugger because the debugger was correctly initializing an otherwise uninitialized variable.
- Scope: On occasion, particularly if the program was large and contained many instantiations, the debugger would not show the source code for a unit because some other unit (one not being stepped through) was not in its active scope. This frequently made it impossible to debug the routine using the debugger.

In spite of these problems, the debugger was a useful tool and was used frequently during CAMP by both the parts and 11th Missile teams.

(3) Configuration Management Tools

The Code Management System (CMS) provided by Digital Equipment Corporation was used for configuration management of the CAMP parts. This tool and its use are further discussed in Section II.2.d.

(4) Documentation Tools

Tools to aid in the creation of top-level and detailed design documents were needed for the following reasons:

- It was anticipated that the top-level and detailed design documents for the CAMP parts would be very large due to the number of CAMP parts and the amount of documentation on each one.
- It was desirable to eliminate the need to maintain three sets of documentation: source code files, top-level design document, and detailed design document. Since all of the information was already contained in the source code files, it was preferable to maintain only them and simply recreate the design documents as necessary.

For these reasons, *comment extractor* tools were developed to help create Section 3.6 (Top-Level Design) of the DoD-STD-2167 Top-Level Design Document and Section 3.3 (Detailed Design) of the DoD-STD-2167 Detailed Design Document. The comment extractors generate the Digital Standard Runoff (DSR) text formatting commands required to produce for the design documents and extract the appropriate information from the source code headers for each of the paragraphs. Figures 10 and 11 show which sections of the source code headers were placed in the design documents.

(5) Miscellaneous Tools/Aids

Naming Convention

A naming convention was established and used for all CAMP files. The primary component of this naming convention was a two-part prefix (i.e., xxx_yyy_). The first part of the prefix (xxx) consisted of the TLCSC identification number (e.g., 621 for Basic_Data_Types, 684 for Geometric_Operations, 001 for Common_Navigation). This part of the prefix was used on all files (e.g., test procedure, test plan, test results) pertaining to a particular TLCSC. The second part of the prefix (yyy) was used to indicate level of nesting of the part contained in the file and was also indicative of compilation order for that TLCSC. This two-part prefix was used for all Ada source code files implementing the TLCSCs.

The use of this naming convention was found to have several benefits. It simplified the use of CMS. For example, by simply specifying "001*.*", a list of all baselined files dealing with the Common_Navigation_Parts TLCSC could be obtained. It also facilitated the development of tools to help with the compilation of parts. This naming convention has now been adopted by several other Ada projects within McDonnell Douglas.

Code Counter

A code counter was developed to help count lines of code and documentation for each of the CAMP parts. The code counter was able to analyze the structure of an Ada source code file and break down the counts among the individual Ada components in the file. For example, the code counter could take the code shown in Figure 9 and tell the user that:

```
Coordinate_Vector_Matrix_Algebra has 2 lines of code and 4 lines of header (not including items nested in it)
Vector_Operations                has 6 lines of code and 4 lines of header (not including items nested in it)
    "+"                          has 2 lines of code and 0 lines of header
    "-"                          has 2 lines of code and 0 lines of header
```

and that:

```
Coordinate_Vector_Matrix_Algebra has 12 lines of code and 8 lines of header (including items nested in it)
Vector_Operations                has 10 lines of code and 4 lines of header (including items nested in it)
```

This tool has proved very useful, both on CAMP and other projects.

```

--*-----
--*TLCSC NAME:
--*  Coordinate_Vector_Matrix_Algebra
--*-----
package Coordinate_Vector_Matrix_Algebra is

--*-----
--*--LLCSC NAME:
--*--  Vector_Operations
--*-----
generic
  type Elements is digits <>;
  type Indices is (<>);
package Vector_Operations is
  type Vectors is array (Indices) of Elements;
  function "+" (Left : Vectors;
                Right : Vectors) return Vectors;
  function "-" (Left : Vectors;
                Right : Vectors) return Vectors;
end Vector_Operations;
end Coordinate_Vector_Matrix_Algebra;

```

Figure 9. Sample Code Counter Input

f. Documentation

All CAMP parts are extensively documented for the following reasons:

- External users of the parts are not familiar with them and therefore need a significant amount of information.
- The CAMP parts make extensive use of generic units, and most users are relatively unfamiliar with the advanced features of generic units. A sample instantiation is included in the code headers of generic parts which shows how other CAMP parts can be used to provide the required generic actual data types, objects, and/or subprograms. In some cases, the sample usage section shows how the generic formal parameters can be used to tailor the part; for example, how to tailor a matrix multiplication routine for use with dynamically sparse matrices. During part development, this portion of the documentation was time-consuming to produce and easily affected by modifications to the part. Later, however, it turned out to be one of the more useful pieces of documentation for the engineers developing the 11th Missile Application since they were unfamiliar with the use of the part.

A Software Development File (SDF) was prepared for all CAMP TLCSCs. Table 4 shows the sections contained in each SDF, along with the information that was maintained in each section.

All of the documentation on a part is contained in its top-level and detailed design headers. A software tool (see Section II.2.e.(4)) was developed to extract information from appropriate sections of the headers for placement in the design documents. Figures 10 and 11 identify the information contained in the CAMP top-level and detailed design headers, and indicate which of these sections are extracted for use in the top-level or detailed design documents.

TABLE 4. SOFTWARE DEVELOPMENT FILE CONTENTS

SECTION	CONTENTS
Requirements	Requirements for this part
Top-level design	Package specification for the TLCSC
Detailed design	Body for the TLCSC
Test plan/procedure	Test procedure/plan for the TLCSC, along with the test code
Test results	Latest set of test results
Problem reports and log	Software discrepancy reports (SDRs) for this TLCSC, along with disposition
Change orders and log	Software enhancement proposal/software change proposal forms (SEP/SCP), along with disposition
Miscellaneous	Walkthrough records

HEADER CONTENTS	EXTRACTED FOR DESIGN DOCUMENT
Name	*
Identification Number	*
Security Level	*
Purpose	*
Requirements trace	*
Context	*
Utilization of external elements	
Packages	*
Subprograms and task entries	*
Exceptions	*
Data types	*
Data objects	*
Input/output	
Generic parameters	
Data types	*
Data objects	*
Subprograms	*
Formal parameters	*
Exported exceptions/types/objects	
Exceptions	*
Data types	*
Data objects	*
Exceptions raised	*
Calling sequence/timing/priority	*
Interrupt handling	*
Sample usage	*
Decomposition	*
Local entities contained in package body	*

Figure 10. Top-Level Design Header Information

The main benefit of using code design headers to produce design documents is that only one set of documentation needs to be maintained. This allows a part to be modified without also modifying documents immediately or trying to remember at a later date which sections of the document need to be updated. When it is time to produce an updated document, the text merely has to be re-extracted. This allows time to produce extensive, high-quality documentation by eliminating tedious and often error-ridden duplication.

<u>HEADER CONTENTS</u>	<u>EXTRACTED FOR DESIGN DOCUMENT</u>
Name	*
Identification Number	*
Security Level	
Purpose	*
Requirements trace	*
Context	*
Utilization of external elements	
Packages	*
Subprograms and task entries	*
Exceptions	*
Data types	*
Data objects	*
Utilization of other elements in top-level component	*
Packages	*
Subprograms and task entries	*
Exceptions	*
Data types	*
Data objects	*
Input/output	
Generic parameters	
Data types	*
Data objects	*
Subprograms	*
Formal parameters	*
Local exceptions/types/objects	
Exceptions	*
Data types	*
Data objects	*
Local entities	*
Exceptions raised	*
Calling sequence	*

Figure 11. Detailed Design Header Information

3. CAMP PARTS PROCESS ANALYSIS

In order to assess productivity for parts development on the CAMP project, effort data was collected from all members of the CAMP team in the areas of domain and requirements analysis, architectural design, detailed design, coding, testing, etc. This was then combined with sizing data to determine productivity. Productivity figures can be misleading, and sometimes impossible to compare because of the many ways they can be calculated. Productivity is generally quoted in terms of lines of code per man-month, but authors frequently don't define terms or specify what is included in code counts.

The size of the CAMP parts was determined using two metrics: lines of code and Ada statements. A line of code was defined as any line in the source code file which contained all or part of an Ada statement. If a single Ada statement occupied three lines in the source code file, then it was counted as three lines of code. A statement count, on the other hand, counted whole Ada statements; in effect counting semicolons. The difference between these two methods of determining code size is illustrated in Figure 12.

The total size of the CAMP parts, in units of lines of code and Ada statements, is shown in Figure 13. As shown in this figure, over 43,000 lines of Ada code were developed during CAMP; this included over 16,000 lines of code for the parts themselves and over 27,500 lines of test code. Using Ada state-

```

package General_Vector_Matrix_Algebra is

  generic
    type Left_Elements      is digits <>;
    type Right_Elements     is digits <>;
    type Output_Elements    is digits <>;
    type Left_Col_Indices   is (<>);
    type Left_Row_Indices   is (<>);
    type Right_Col_Indices  is (<>);
    type Right_Row_Indices  is (<>);
    type Output_Col_Indices is (<>);
    type Output_Row_Indices is (<>);
    type Left_Matrices is array (Left_Row_Indices,
                                Left_Col_Indices) of Left_Elements;
    type Right_Matrices is array (Right_Row_Indices,
                                Right_Col_Indices)
                                of Right_Elements;
    type Output_Matrices is array (Output_Row_Indices,
                                Output_Col_Indices)
                                of Output_Elements;
    with function "*" (Left : Left_Elements;
                     Right : Right_Elements)
                     return Output_Elements is <>;
  package Matrix_Matrix_Transpose_Multiply_Unrestricted is

    function "*" (Left : Left_Matrices;
                 Right : Right_Matrices) return Output_Matrices;

  end Matrix_Matrix_Transpose_Multiply_Unrestricted;

end General_Vector_Matrix_Algebra;

```

Counting the above code, using lines of code and Ada statements as the metrics, yields the following results:

<u>Lines of Code</u>	<u>Ada Statements</u>
27	16

Figure 12. Lines of Code versus Ada Statements

ments as the sizing metric, over 28,000 Ada statements were developed during CAMP with over 10,000 of these being part code and almost 18,000 being test code.

SIZE

	LINES OF ADA CODE	ADA STATEMENTS	LINES OF COMMENTS
PART CODE	16,091	10,203	91,553
TEST CODE	27,584	17,991	
TOTAL	43,675	28,194	

Figure 13. CAMP Parts Sizing Data

On any software project, source code must be developed and documented. Section II.2.f discusses the vital role extensive documentation plays in the successful use of reusable software. This is reflected in the sizing data contained in Figure 13 which shows that the ratio of lines of comments to lines of code is approximately 5.7:1 and the ratio of lines of comments to Ada statements is almost 9:1.

Code size is not the only factor in determining productivity; effort must also be assessed. Effort data for development of the CAMP parts is shown in Figure 14. It includes the number of hours expended for all phases of the CAMP parts life cycle, from domain analysis through maintenance. A total of 9734 man-hours of effort went towards the development of the CAMP parts, with 6557 of these hours expended during the design and testing phases.

EFFORT	
	ACTUALS AT COMPLETION
DOMAIN ANALYSIS	1153
REQUIREMENTS SPEC.	1428
DESIGN	4010
TEST PLANNING	1334
CODING	516
TESTING	697
MAINTENANCE	596
TOTAL	
9734	
DESIGN-TESTING	
6557	
(IN MAN-HOURS)	

Figure 14. CAMP Parts Effort Data

The productivity statistics for development of the CAMP parts, using several metrics, is shown in Figure 15. The importance of knowing how productivity is being measured can be seen in this figure which shows that productivity figures from 164 statements/man-month to 1039 lines of code/man-month can be justified, depending on how and what code is counted and what is included in the man-month figures.

Figure 16 gives an overall picture of the development statistics for the CAMP parts development effort. The data includes the most conservative numbers shown in Figure 15, using code counts for parts code only and man-month figures for the entire CAMP effort. It can be seen from this figure that the productivity experienced during CAMP parts development was approximately 61% greater than that predicted by COCOMO for embedded software development. Several factors contributed to this increased productivity:

- Ada: The Ada language itself contributes to increased productivity. Strong data typing, for example, helps to ensure that many errors are found during compilation rather than being found during testing when they would be more time consuming and costly to correct.
- Good people: All members of the CAMP parts development team had at least some Ada experience prior to joining the project, and all received training in software engineering practices

PRODUCTIVITY			
DESIGN-TESTING EFFORT	PART CODE ONLY	PART & TEST CODE	
	LOC/MM 383 STMT/MM 243 MH/LOC 0.407 MH/STMT 0.643	LOC/MM 1039 STMT/MM 671 MH/LOC 0.150 MH/STMT 0.233	
ALL EFFORT	PART CODE ONLY	PART & TEST CODE	
	LOC/MM 258 STMT/MM 164 MH/LOC 0.605 MH/STMT 0.954	LOC/MM 750 STMT/MM 452 MH/LOC 0.223 MH/STMT 0.345	

(156 MH/MM)

Figure 15. CAMP Parts Productivity Data

SIZE	16,091 LOC	10,203 STMTS
PRODUCTIVITY	258 LOC/MM	164 STMTS/MM
EXPECTED PRODUCTIVITY (COCOMO)	160 LOC/MM	
DELTA	61% ↑	• ADA • GOOD PEOPLE • GOOD TOOLS • REUSE

Figure 16. CAMP Parts Development Statistics

either before or after joining the project. In addition, several members of the team had extensive Ada experience and were available to help train new people. There was also continuity of personnel between CAMP-1 and CAMP-2, with key members of the CAMP-1 team remaining throughout CAMP-2. This provided increased consistency in the overall design philosophy of the parts and increased the ability to pass on the lessons learned during earlier phases of development.

- Good tools: As discussed in Section II.2.e, various tools were used throughout the CAMP program; they made a significant contribution to increased productivity.

- **Code reuse**: During CAMP, not only was reusable code developed, but it was used in the development of more reusable code. It was often possible to use previously developed specifications and/or bodies to create new reusable parts. A simple example of this involves matrix addition and subtraction routines. Since the differences between the two are minor, the matrix addition routine can be developed and completely documented, and then the subtraction routine can be created simply by copying and making minor modifications to the addition routine.

SECTION III

INTER-RELATIONSHIPS BETWEEN CAMP PARTS

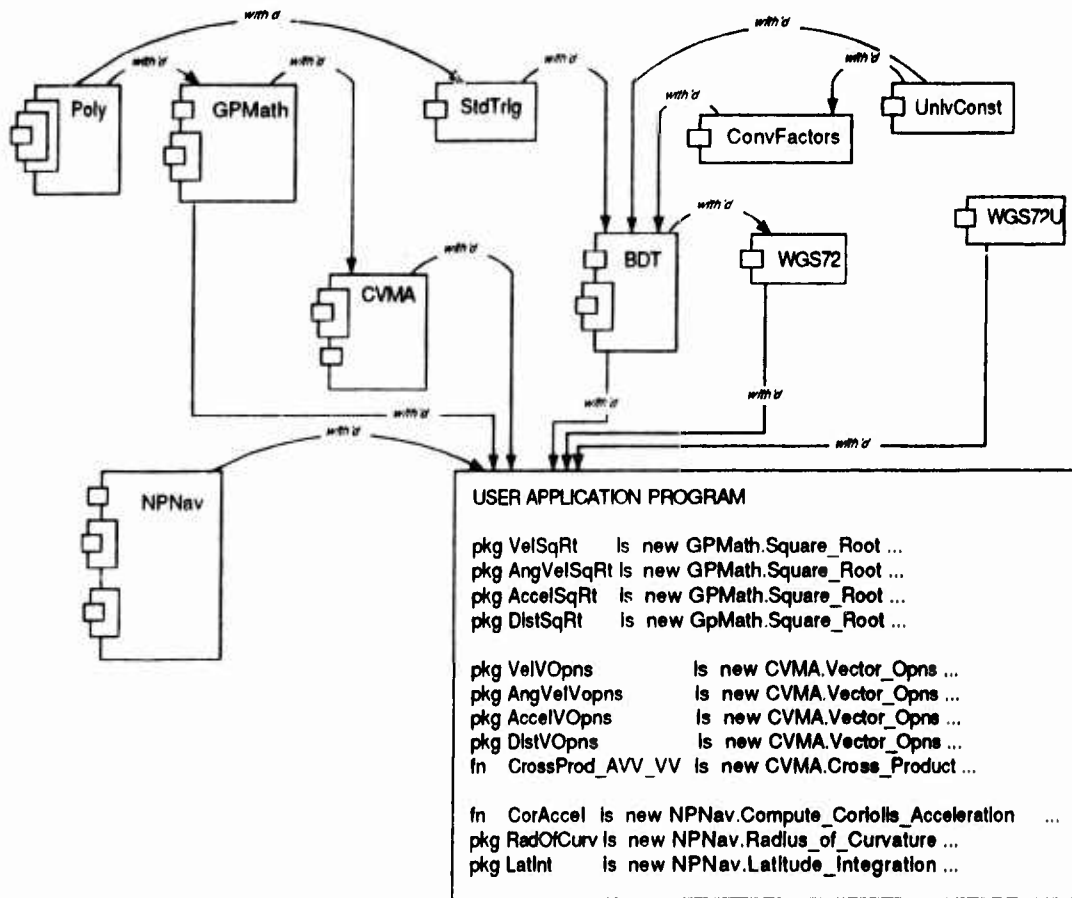
An important aspect of the design of the CAMP parts is the way various parts were designed to build on other parts, work together, and facilitate using other parts. These relationships between the parts are further discussed in the following paragraphs. Figure 17 shows how these relationships come into play when developing a small portion of a navigation system.

1. PARTS BUILD ON OTHER PARTS

One example of parts building on other parts involves the `Polynomials`, `Standard_Trig`, and `Basic_Data_Types` TLCSCs as illustrated in Figure 18. The `Polynomials` TLCSC lies at the bottom of the build and provides an extensive set of polynomial solutions to various transcendental functions. The generic `Standard_Trig` TLCSC forms the second layer by exporting trigonometric data types and operations. `Standard_Trig` uses the `Polynomials` package to obtain the required polynomial solutions to its exported transcendental functions. The `Basic_Data_Types` TLCSC provides the final layer. In addition to providing a set of data types and operations typical of a navigation implementation, `Basic_Data_Types` instantiates the `Standard_Trig` package. This design approach offers several advantages:

- Minimal functionality is added from one step to the next.
- Users of the higher level packages, such as `Basic_Data_Types`, frequently will not need to reference the lower level packages, such as `Polynomials`.
- Finally, combining the parts saves work for the user.

In this example, a user merely needs to import `Basic_Data_Types` in order to obtain a full set of navigation data types (such as various forms of distances, velocities, accelerations, etc.), operators upon these types, trigonometric data types (such as radians, degrees, etc.), and a full set of trigonometric functions.



1. A total of 10 packages must be compiled into the user's library. The user himself requires six of these (indicated by arrows into the user application); the six packages require an additional four.
2. The user must do the following before instantiating the navigation parts:
 - Instantiate four versions of the square root package (GPMath.Square_Root) using data types and operators supplied by the basic data types (BDT) package.
 - Instantiate four versions of the vector operations package (CVMA.Vector_Opns) using data types and operators supplied by BDT and the square root functions contained in the packages previously instantiated by the user.
 - Instantiate a cross product function using scalar data types and operations supplied by BDT, along with vector data types and operations obtained from three separate instantiations of CVMA.Vector_Opns.
3. The three navigation parts can then be instantiated using:
 - Scalar data types and operators supplied by BDT.
 - Scalar data types and trigonometric functions supplied by an instantiation of the standard trig package contained in BDT (BDT.Trig).
 - Vector types and operations supplied by the four instantiations of CVMA.Vector_Opns.
 - Data constants supplied by the WGS72 ellipsoid metric data package (WGS72) and the WGS72 ellipsoid unitless data package (WGS72U).
 - User-defined data types and objects.

Figure 17. Assembling a North-Pointing Navigation System

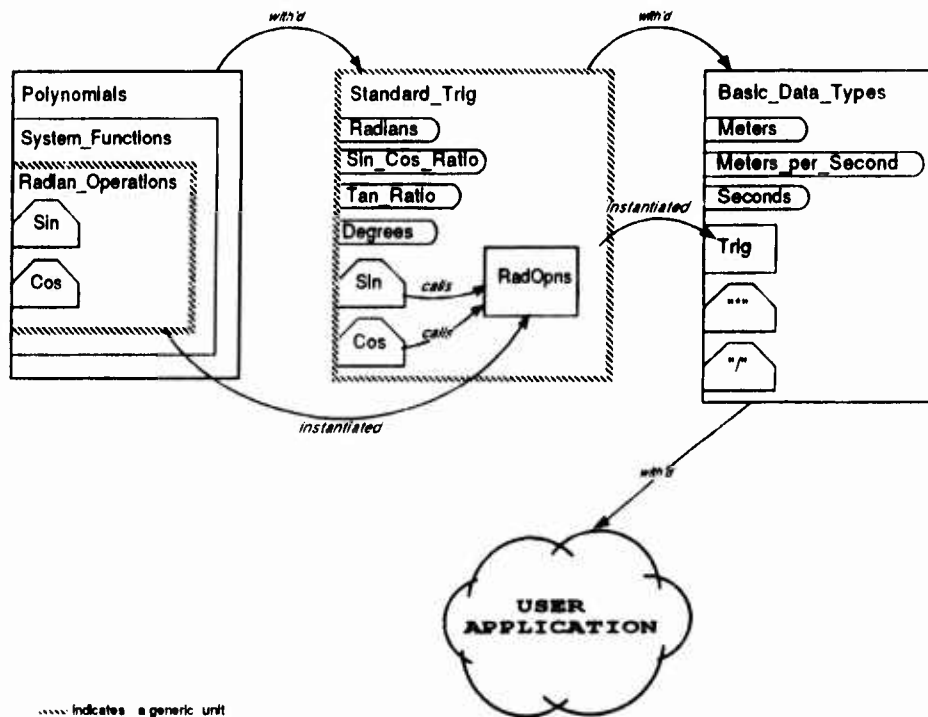


Figure 18. Some Parts Build On Other Parts

2. PARTS WORK TOGETHER

Parts were also designed to work together, using low-level parts to support more complex operations. This design approach differs from the approach previously discussed in that functionality is added with each step and the lower TLCSCs are frequently required by the user. An example of this interrelationship can be seen in the Geometric_Operations and Waypoint_Steering TLCSCs shown in Figure 19. The Waypoint_Steering TLCSC exports the Steering_Vector_Operations package which handles the initialization and updating of waypoint steering vectors. In order to perform its operations, the Steering_Vector_Operations package instantiates two subroutines from the Geometric_Operations package which are designed to calculate unit radial vectors, unit normal vectors, and course segments. This design methodology has several benefits:

- Since the geometric operations are not placed in the package body of the Waypoint_Steering TLCSC, they are also available to the user.
- Not duplicating the Geometric_Operations code within the Waypoint_Steering TLCSC improves maintainability.
- Performing the instantiations of the Geometric_Operations parts within the Steering_Vector_Operations LLCSC instead of bringing them in as generic subroutines saves the user the work of finding the additional parts and doing the instantiations.

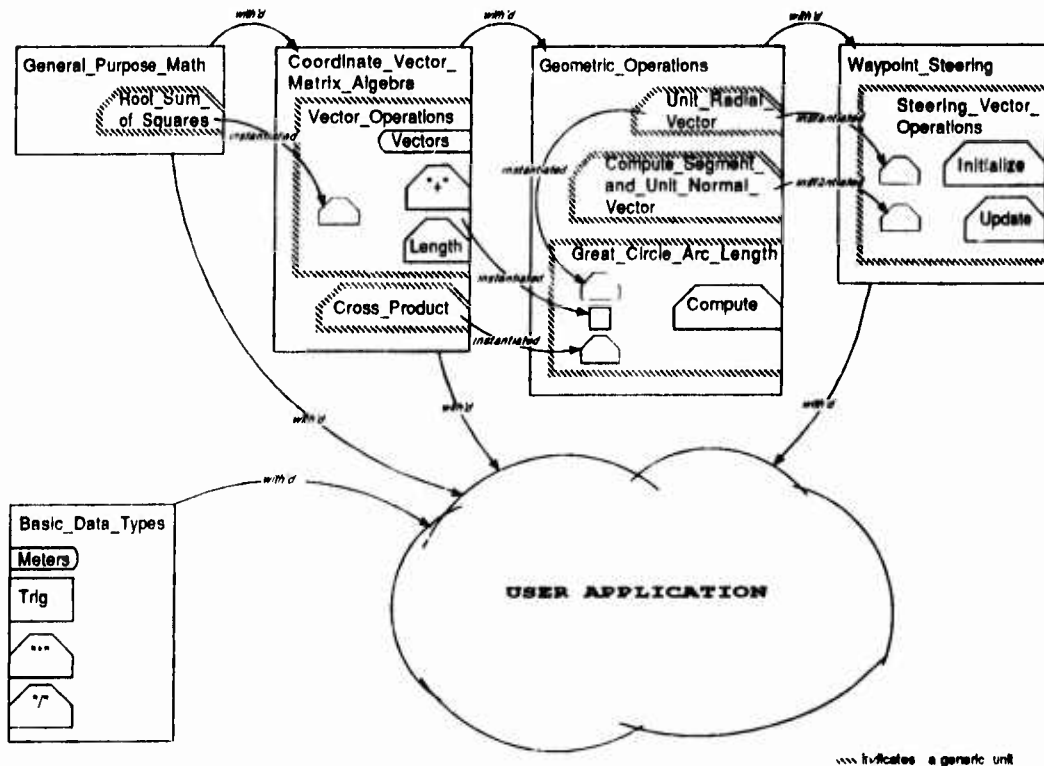


Figure 19. Parts Work Together

3. CAMP PARTS FACILITATE USE OF OTHER PARTS

Finally, parts were designed to facilitate using other parts by providing the requisite generic actual parameters. An example of this is shown in Figure 20. In order to instantiate the generic `Compute_Segment_and_Unit_Normal_Vector` procedure, the only data type the user needs to define is a discrete type for `Indices`. The remaining scalar types can be obtained from the `Basic_Data_Types` package, along with the multiplication and division operators; the vector type and operations on that type (i.e., `Vector_Length` and `Cross_Product`) can be obtained by instantiating the `Vector_Operations` package in the `Coordinate_Vector_Matrix_Algebra` TLCSC; and a value for the radius of the Earth can be found in the `WGS72_Ellipsoid_Engineering_Data` TLCSC. This kind of support can be found in most of the CAMP parts.

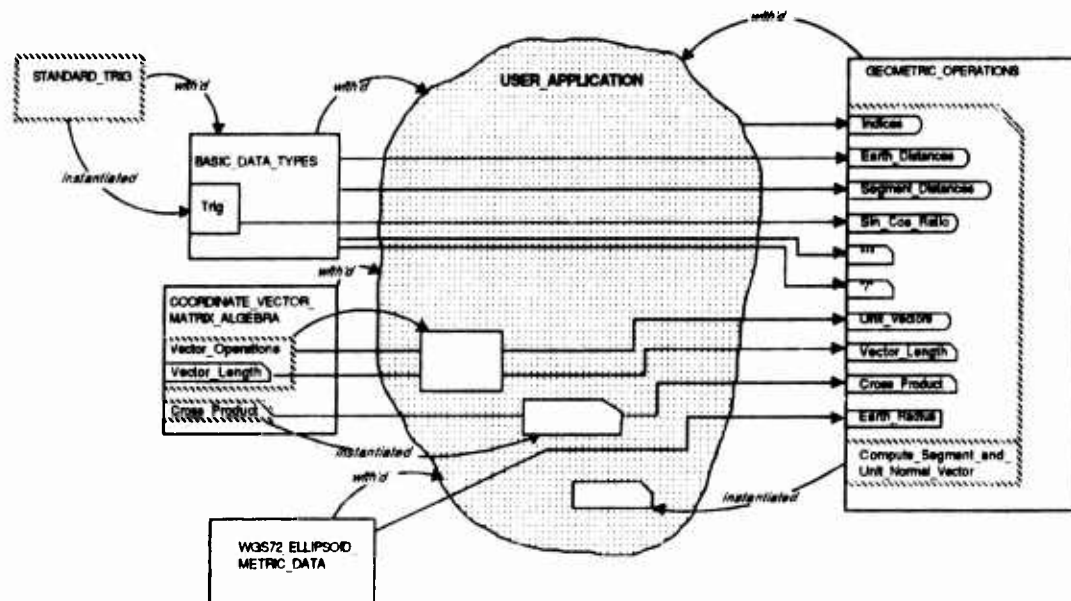


Figure 20. CAMP Parts Facilitate Use of Other Parts

When designing the CAMP parts, a primary consideration was how to provide low-level operations, such as linear algebra and transcendental functions, to the more complex routines. There were several options:

1. In-line the required operations directly into the higher level routine: This option was considered unacceptable since it would have caused the parts to become excessively large. Also, in-lining would have increased testing time and brought about the potential for a maintenance nightmare.
2. Place the required code in subroutines located in package bodies: This option, while an improvement over option 1, would also increase the size of the parts, lengthen testing time, and increase maintenance difficulties.
3. Instantiate a required operation from another CAMP part: In a few cases, this option was chosen. This method was considered desirable if: 1) only one method existed for implementing the required operation; or 2) the instantiating part were a very high-level part, such as a Kalman update package, designed to provide one possible solution to a problem by bringing together one possible combination of lower level parts.

This option was not considered acceptable if the required operation was a very basic one, such as a trigonometric function, and there was no way of knowing ahead of time which algorithm would provide optimal performance.

4. Bring in the required operations via generic parameters: This option was chosen in the vast majority of cases.

The use of generic formal subprograms to import required operations is an important design feature

of the parts. It has the advantage of providing great flexibility to the user by providing CAMP parts to supply low level operations or allowing the user to define his own, as shown in the following examples.

• Example 1

In this example, assume the user wishes to instantiate both of the parts contained in the Geometric_Operations TLCSC shown in Figure 21. Each part requires a sine/cosine procedure as a generic parameter. If the user has imported the Basic_Data_Types (BDT) package, he already has access to the sine/cosine procedure provided indirectly by BDT's instantiation of Standard_Trig (Trig). If this procedure is satisfactory for his computations, the user need not specify it in his instantiation since the BDT version will be selected by default. If, however, the user feels his calculations require more accuracy or speed, he may construct a different sine/cosine procedure by building one from the over 25 sine functions provided by the Polynomials TLCSC or by writing his own. This new sine/cosine procedure may then be used in one of the following ways:

- If he wishes to use this new procedure throughout his application for all sine/cosine calculations, the procedure can be specified in such a way as to hide the sine function contained in BDT.Trig. He can then let the generic actual subroutines default to this new procedure. This is illustrated in Figure 22.
- If the newly created sine/cosine procedure is to be used only for certain calculations, it can be designed in such a way as to not hide the one contained in BDT.Trig. In this case, the special procedure would have to be explicitly specified in instantiations where it was to be used. Using this method, it is possible for the user to create multiple sine/cosine procedures — a fast one, a highly accurate one, and a general purpose one — to meet his needs. This is illustrated in Figure 23.

```

generic
  type Angle      is digits <>;
  type Trig_Ratio is digits <>;
package Standard_Trig is

  type Radians      is new Angle;
  type Sin_Cos_Ratio is new Trig_Ratio range -1.0..1.0;

  procedure Sin_Cos (Input      : in    Radians;
                     Sin_Result : out Sin_Cos_Ratio;
                     Cos_Result : out Sin_Cos_Ratio);

end Standard_Trig;

-----

with SYSTEM;
with Standard_Trig;
package Basic_Data_Types is

  type Real      is digits SYSTEM.MAX_DIGITS;
  type Meters    is digits SYSTEM.MAX_DIGITS;

  package Trig is new Standard_Trig
    (Angle      => Real,
     Trig_Ratio => Real);

  type Earth_Position_Radians is new Trig.Radians;

  function "*" (Left  : Meters;
               Right : Trig.Sin_Cos_Ratio)
    return Meters;

end Basic_Data_Types;

```

```

package Geometric_Operations is

generic
  type Indices      is (<>);
  type Earth_Positions is digits <>;
  type Sin_Cos_Ratio is digits <>;
  type Unit_Vectors is array (Indices)
    of Sin_Cos_Ratio;

  X : in Indices      := Indices'FIRST;
  Y : in Indices      := Indices'SUCC(X);
  Z : in Indices      := Indices'LAST;
  with procedure Sin_Cos
    (Input      : in    Earth_Positions;
     Sine       : out Sin_Cos_Ratio;
     Cosine     : out Sin_Cos_Ratio)
    is <>;

  function Unit_Radial_Vector
    (Lat_of_Point : Earth_Positions;
     Long_of_Point : Earth_Positions)
    return Unit_Vectors;

generic
  type Earth_Distances is digits <>;
  type Earth_Positions is digits <>;
  type Segment_Distances is digits <>;
  type Sin_Cos_Ratio is digits <>;
  Earth_Radius      : in Earth_Distances;
  with function "*" (Left  : Earth_Distances;
                   Right : Sin_Cos_Ratio)
    return Segment_Distances is <>;
  with function Sqrt (Input : Sin_Cos_Ratio)
    return Sin_Cos_Ratio is <>;
  with procedure Sin_Cos
    (Input      : in    Earth_Positions;
     Sine       : out Sin_Cos_Ratio;
     Cosine     : out Sin_Cos_Ratio)
    is <>;

package Great_Circle_Arc_Length is

  function Compute
    (Latitude_A : Earth_Positions;
     Latitude_B : Earth_Positions;
     Longitude_A : Earth_Positions;
     Longitude_B : Earth_Positions)
    return Segment_Distances;

end Great_Circle_Arc_Length;

end Geometric_Operations;

```

Figure 21. Required Operations Obtained Through Use of Generic Formal Parameters

```

with Basic_Data_Types;
with Geometric_Operations;
with WGS72_Ellipsoid_Metric_Data;
procedure User_Application is

    use Basic_Data_Types;

    package BDT renames Basic_Data_Types;
    package GEO renames Geometric_Operations;
    package WGS72 renames WGS72_Ellipsoid_Metric_Data;

    type Indices is (X, Y, Z);

    type Unit_Vectors is array (Indices) of BDT.Trig.Sin_Cos_Ratio;

    function Sqrt (Input : BDT.Trig.Sin_Cos_Ratio)
        return BDT.Trig.Sin_Cos_Ratio;

-- --new Sin_Cos procedure to override that provided by BDT.Trig
    procedure Sin_Cos (Input : in      BDT.Earth_Position_Radians;
                       Sine   : out   BDT.Trig.Sin_Cos_Ratio;
                       Cosine : out   BDT.Trig.Sin_Cos_Ratio);

    function U_Radial_Vector is new GEO.Unit_Radial_Vector
        (Indices           => Indices,
         Earth_Positions   => BDT.Earth_Position_Radians,
         Sin_Cos_Ratio     => BDT.Trig.Sin_Cos_Ratio,
         Unit_Vectors      => Unit_Vectors);

-- --Sin_Cos defaults to new Sin_Cos procedure

    package Great_Circle_Arc_Len is new GEO.Great_Circle_Arc_Length
        (Earth_Distances   => BDT.Meters,
         Earth_Positions   => BDT.Earth_Position_Radians,
         Segment_Distances => BDT.Meters,
         Earth_Radius      => WGS72.Earth_Equatorial_Radius,
         Sin_Cos_Ratio     => BDT.Trig.Sin_Cos_Ratio);

-- --Sin_Cos defaults to new Sin_Cos procedure

begin
    ...
end User_Application;

```

**Figure 22. Sample Instantiations of Geometric_Operations Parts
Using Default Routines**

```

with Basic_Data_Types;
with Geometric_Operations;
with WGS72_Ellipsoid_Metric_Data;
procedure User_Application is

    use Basic_Data_Types;

    package BDT renames Basic_Data_Types;
    package GEO renames Geometric_Operations;
    package WGS72 renames WGS72_Ellipsoid_Metric_Data;

    type Indices is (X, Y, Z);

    type Unit_Vectors is array (Indices) of BDT.Trig.Sin_Cos_Ratio;

    function Sqrt (Input : BDT.Trig.Sin_Cos_Ratio)
        return BDT.Trig.Sin_Cos_Ratio;

-- --additional Sin_Cos procedure
    procedure Fast_Sin_Cos (Input : in BDT.Earth_Position_Radians;
        Sine : out BDT.Trig.Sin_Cos_Ratio;
        Cosine: out BDT.Trig.Sin_Cos_Ratio);

    function U_Radial_Vector is new GEO.Unit_Radial_Vector
        (Indices => Indices,
         Earth_Positions => BDT.Earth_Position_Radians,
         Sin_Cos_Ratio => BDT.Trig.Sin_Cos_Ratio,
         Unit_Vectors => Unit_Vectors,
         Sin_Cos => Fast_Sin_Cos);

    package Great_Circle_Arc_Len is new GEO.Great_Circle_Arc_Length
        (Earth_Distances => BDT.Meters,
         Earth_Positions => BDT.Earth_Position_Radians,
         Segment_Distances => BDT.Meters,
         Earth_Radius => WGS72.Earth_Equatorial_Radius,
         Sin_Cos_Ratio => BDT.Trig.Sin_Cos_Ratio);

-- --Sin_Cos defaults to BDT.Trig.Sin_Cos

begin
    ...
end User_Application;

```

Figure 23. Sample Instantiations of Geometric_Operations Parts
Using Specialized Sin_Cos Procedure

- **Example II**

In this example, the user wishes to construct a Kalman filter using a complicated-H matrix. If he uses the `Kalman_Filter_Data_Types` package and all the data types it provides, all generic formal subroutines required by instantiations of any of the parts contained in the `Kalman_Filter_Common_Parts` and `Kalman_Filter_Complicated_H_Part` TLCSCs will properly default. If however, he wishes to use reduced storage rather than full storage matrices, it is possible for him to define his own data types and operations and still use the Kalman filter parts without making any modifications to the parts themselves. This latter option is the one that was chosen for the 11th Missile Application.

SECTION IV

DEVELOPMENT AND TESTING OF A PARTS COMPOSITION SYSTEM (PCS)

The major problems associated with software reuse efforts have been the lack of information on the availability and applicability of reusable parts and the lack of information on how to use those parts. During the CAMP-1 feasibility study, it was concluded that software reuse would not come to fruition if there were not some mechanism for assisting the potential user in identifying, locating, and using available software parts. One such mechanism is a parts composition system (PCS) which can facilitate the use of existing software parts by providing tools to perform some of the mechanical tasks associated with software reuse.

The objective of the CAMP-1 feasibility study, with respect to parts composition systems, was to determine the feasibility and value of automating some, or all, of the process of using and managing software parts. The study involved an investigation of both short and long-term possibilities. Feasibility was clearly established (Reference 7), and the requirements and top-level design of a parts composition system were specified during CAMP-1.

During CAMP-2, a prototype parts composition system was implemented and tested, and then used by the 11th Missile development team to demonstrate its utility and value. This prototype, which is referred to as the Ada Missile Parts Engineering Expert (AMPEE) system, alleviates many of the problems associated with software reuse by providing the user with an expert assistant to advise him on the availability and relevance of CAMP reusable Ada software parts to his application, and to aid in the development of software systems by automatically generating the required code for particular operations or subsystems of the application, e.g., navigation, Kalman filter, or autopilot operations.

1. PCS FUNCTIONALITY

Although much of the AMPEE system is CAMP-specific, the underlying principles are applicable to a variety of domains. The AMPEE system established the functions required of a parts composition system to assist the user in using reusable software parts.

A three-pronged approach was taken in assisting the user with the reusable CAMP software parts. This approach is embodied in the three major subsystems of the AMPEE system — Parts Catalog, Parts Identification, and Component Construction. The Parts Catalog subsystem is similar to an automated card catalog for books, i.e., it is used to locate reusable software parts and obtain information about those parts. This subsystem also provides a means to maintain the catalog in an up-to-date form. The Parts Identification subsystem provides the user with access to the on-line parts catalog at a very high level. Unlike the Parts Catalog subsystem which requires the user to have some idea of the types of parts that he is looking for, the Parts Identification subsystem provides the user with access to the information in the catalog based solely on his knowledge of his own application, i.e., before he knows about specific parts. The Component Construction subsystem provides the user with a means of generating tailored Ada com-

ponents based on reusable meta-parts that are in the Parts Catalog. Meta-parts were described in the CAMP-1 Final Technical Report 7, and are discussed further in Section IV.1.c. Each of these subsystems is discussed in greater detail in the following paragraphs.

a. Parts Catalog

The backbone of the AMPEE system is a software parts catalog for the CAMP reusable Ada missile software parts. Earlier research (during the CAMP feasibility study) indicated that a major limiting factor in the widespread acceptance and use of off-the-shelf software was the lack of reliable information describing the parts in adequate detail to determine their applicability to a particular software project. Under the CAMP project, a catalog was developed that provides the type of information that is needed to make informed decisions about parts. Each reusable software part is described by numerous attributes; these are enumerated in Figure 24, and described in detail in Appendix B.

GENERAL	
Part Number	Revision Number
Part Name	Functional Abstract
Mode	Taxonomic Category
Class	Keywords
Last Change Date of Entry	Project Usage
Government Security Classification (part)	Corporate Sensitivity Level (part)
Government Security Classification (entry)	Corporate Sensitivity Level (entry)
Remarks	
DEVELOPMENT	
Design Issues	Revision Notes
Development Date	Developer
Development Status	Developed For
Requirements Documentation	Design Documentation
USAGE	
Location of Source Code	Access Notes
Withs	Withed By
Implemented By	Implements
Built From	Used to Build
Sample Usage	Hardware Dependencies
Restrictions	
PERFORMANCE	
Source Size/Complexity Characterizations	Fixed Object Code Size
Timing	Accuracy

Figure 24. Catalog Attributes

It is important to distinguish between the CAMP parts themselves and the software entities that are cataloged. Parts were defined in Section II.1. There is not a one-to-one correspondence between

CAMP parts and catalog entries. Although parts are cataloged, Ada package bodies are cataloged separately from their specifications; encapsulating packages are also cataloged. Thus, although approximately 450 CAMP Ada parts have been implemented and tested to date, there are over 1100 catalog entries. An examination of the catalog attribute *class* provides a clearer distinction between parts and catalog entries. The *class* attribute identifies the type of entity that can be cataloged; it encompasses software entities such as package specifications, package bodies, generic task specifications, generic task bodies, generic formal parts, and context clauses.

It should be noted that there is a hardcopy form of the CAMP software catalog as well as the on-line version that is incorporated into the AMPEE Parts Catalog subsystem. The hardcopy form is useful for those who do not have access to the AMPEE system. The on-line version provides specific information on available reusable software parts from within the AMPEE system.

(1) Design

The AMPEE Parts Catalog subsystem allows a user of the AMPEE system to access and maintain the CAMP parts catalog entries. Maintenance functions include functions to add entries for new or revised reusable software entities, and to modify or delete entries. Locating functions include functions to search for catalog entries based on various attribute values, examine both catalog entries and Ada part source code, and to generate printed versions of the catalog entries. Catalog interaction is carried out via a structured dialog between AMPEE and the user; the user provides all information necessary for the system to implement his catalog request. Figure 25 depicts the functions that comprise the Parts Catalog subsystem.

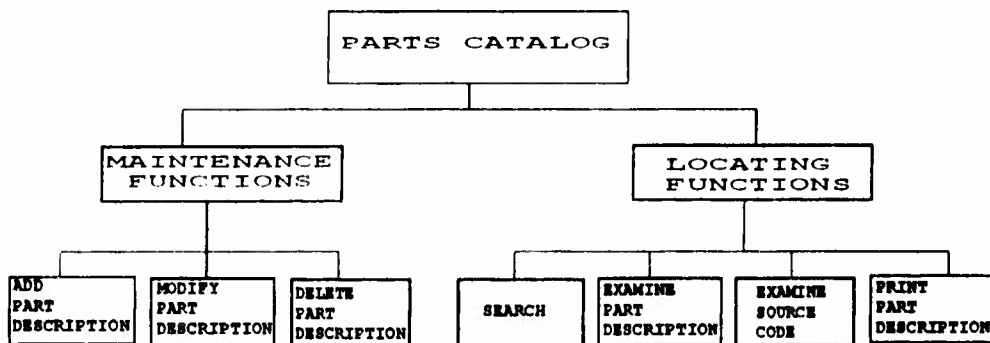


Figure 25. Parts Catalog Functions

For operations that can be performed on an existing catalog entry, the user can provide a specific part id, request a menu of all part ids, or request a menu of part ids in the current *search list*. The *search list*, if it exists, is a list of the part ids that have satisfied the search criteria specified during the most recent search operation, or have been returned by one of the Parts Identification functions.

The Add Part Description function allows the user to add an entry to the CAMP parts catalog for a new or revised CAMP software part. This can be done in one of three ways:

- A new part description of a new part is entered (i.e., "from scratch")

- A new part description for a revision of an existing part is entered
- A new part description of a new part is entered by copying the part description of an existing part and modifying it as needed.

A unique *part id* is generated for each part that is entered. The part id consists of a part number and a revision number, and is not intended to have any semantic meaning. The user is led through the addition of required and recommended attributes for each part entry that is added to the catalog. *Required* attributes are those which have been deemed to be essential in providing the catalog user with sufficient information to make an informed decision as to the appropriateness of a given CAMP Ada part. Required attributes are enumerated in Figure 26. Two additional attributes, *withs* and *withed by*, are defined as required, but because they may not always be applicable, it is the user's responsibility to provide them. *Recommended* attributes are those that, although they provide useful information, are not usually critical to making a determination as to the appropriateness of a part.

The AMPEE system provides the values of some attributes such as the revision number, date of change of the catalog entry, and values for inverses (e.g., if the user enters *built from* data, the system will automatically update the appropriate other catalog entries with *used to build* data). Other attribute values must be explicitly provided by the user.

Part Number
Revision Number
Part Name
Taxonomic Category
Functional Abstract
Class
Mode
Last Change Date of Entry
Development Date
Developer
Development Status
Government Security Classification of Part
Government Security Classification of Entry
Corporate Sensitivity Level of Part
Corporate Sensitivity Level of Entry

Figure 26. Required Catalog Attributes

The Modify Part Entry function allows the user to modify an existing entry in the CAMP software parts catalog. After indicating which part entry is to be modified, the user is allowed to select the attributes that are to be modified. He then provides the system with the new data so that the catalog entry can be updated.

The Delete Part Entry function allows the user to delete an entry from the CAMP software parts catalog. The user must indicate which part entry is to be deleted; that entry is then deleted from the current catalog. In order for the deletion to be permanent, the user, upon exiting the AMPEE system, must indicate that all catalog changes made during the current session are to be saved.

The Search function allows the user to explore the reusable software parts that are available to him. Inquiry can take place along a number of lines (e.g., keywords or other attributes), and multiple selection criteria are supported.

For the keyword search, the user must identify the keywords and/or phrases that are to be used as the selection criteria. Within the parts catalog, keywords are generally entered for high-level parts only (this reduces the number of parts that will be returned by a search, thus making it more meaningful); other attributes, such as *built from*, can be used to obtain related parts. For the searches on other attributes, the user must identify both the attribute name and value to be used as the selection criteria. The searchable attributes are enumerated in Figure 27.

If any matches are found during the search, their part ids are displayed for the user, and the list of part ids for the *matches* is kept for further manipulation. The user can specify further search criteria to be applied to the parts in the search list, or he can select part ids from the list for further processing (e.g., deletion, examination of catalog entry or source code). If no matches are found, then a message is displayed indicating this.

Part Name
Mode (Bundled, Unbundled, or Schematic)
Taxonomic Category
Class
Government Security Class of Part
Government Security Class of Entry
Corporate Sensitivity Level of Part
Corporate Sensitivity Level of Entry
Project Usage
Last Change Date of Entry
Implements
Implemented By
With
Withed By
Built From
Used to Build
Location of Source Code
Developer
Developed For
Development Date
Development Status

Figure 27. Searchable Catalog Attributes

The Examine Part Description function allows a user to retrieve and examine a catalog entry for a specified part in the CAMP parts catalog. The user must identify the part entry that is to be examined. He can then view the basic attributes (i.e., part id, name, last change date of entry, development date, development status, developer, mode, class, taxonomic category, and government and corporate sensitivity levels of the part and part entry), or select additional attributes to view.

The cataloged software parts are classified in part by their *mode* (i.e., whether they are bundled, unbundled, or schematic parts); Appendix B describes this attribute in more detail. The Examine Source Code function allows the AMPEE system user to examine the actual source code for reusable CAMP parts that are classified as either bundled or unbundled parts. Schematic parts cannot be examined because there is no actual source code until a component is constructed via the AMPEE system.

The Print Catalog Entry function provides the user with the ability to obtain a formatted hardcopy of one or more catalog entries. The user can process the entire catalog, entries obtained from a search list, or individually identified entries. Output may be sorted in ascending order by part id or alphabetically by taxonomic category. The user has two options in directing the output: he can have it

print to both the screen and to a file, or just to a file. Formatting is performed via the text processing program Scribe. Because of limitations of the Scribe system, it is not possible to view text interactively after it is formatted by the Scribe processor; thus the output displayed on the screen is *not* identical to that produced for printing.

(2) Testing and Operational Evaluation

The AMPEE Parts Catalog subsystem underwent several levels of testing:

- Testing by the subsystem developer
- Use for entry of catalog data
- Use by the AMPEE system training class

Testing was performed by the subsystem developer to eliminate both programming errors, and interface errors or inconsistencies. Although this type of testing is important, it cannot uncover all of the problems that may exist.

The Parts Catalog subsystem was used for the entry of data into the catalog. This data entry was performed by a number of persons with varying backgrounds, including a high school student with no previous exposure to the system; a college student with no software engineering training, a member of the PCS development team who had not worked on this particular subsystem, and the senior member of the parts development team. All of these users were able to successfully use the system with very little instruction, and some with very little background knowledge of the project itself. Although these users were able to easily pick up the knowledge needed to perform data entry, they did uncover inconsistencies in the interface, and highlighted some areas for improvement. As a result of this use, several additional subfunctions were added (e.g., add new part by copying existing entry).

This subsystem was also used by the AMPEE system training class for instructional purposes. These users also found the interaction to be relatively straightforward, but they also uncovered several inconsistencies and a few minor errors that had not been previously identified.

Overall, the AMPEE Parts Catalog subsystem was found to be useful, although several problems were identified. These were mostly a result of the prototype nature of the system, and included items such as response time and start-up time. Some users also thought the system could be improved by providing greater *carry-over* between the functions within the AMPEE system.

b. Parts Identification

The Parts Identification subsystem provides the user with two capabilities for determining the availability of potentially applicable parts for a given software system; the functions map software requirements to software parts. The user, who would generally be a missile system engineer or a missile software requirements engineer, can provide the system with his requirements and determine the parts that may be applicable to his project. Although the Parts Catalog subsystem also provides information on the potential applicability of parts, the Parts Identification functions provide this information at a higher level, i.e., the user does not need to know about specific parts to obtain information; he need only provide information about his application.

The Parts Identification functions are intended for use early in the development cycle — as early as the missile system requirements/design phase, or in the pre-software development phase. Their use this early can help drive the design in a direction that can make maximum use of existing software. If software designers wait until after the requirements and design phase to start exploring options for reuse of existing software, it is generally too late. At that point, the design may be such that certain parts are excluded from reuse. In addition to driving the design, the Parts Identification functions can also be used to facilitate software cost estimates, sizing and timing studies, and *make-or-buy* trade-off studies.

During the CAMP study, two approaches to software parts identification were identified. One is an *application* approach, that looks at overall system requirements. By viewing the system as a whole, the user can see the effect of various trade-offs in algorithms. Consistency can also be provided by ensuring that parts identified for the user are not incompatible. The *architectural* approach looks at the subsystems that are needed in a particular missile system. This approach is based on hierarchical models of missile flight software. The user provides information on his application and a model is presented for his viewing. He can then see the subsystems and parts may be needed. Both approaches have been incorporated into the AMPEE system — the application approach is embodied in Application Exploration, and the architectural approach is embodied in Missile Model Walkthrough.

(1) Application Approach

The application approach to parts identification is embodied in the Application Exploration function within the AMPEE system. Application Exploration provides the user with the capability of mapping high-level requirements to available software parts. It is intended for use early in a software development project to identify software parts with potential applicability to the user's current needs. The user is asked a number of questions about his application, and a list of potentially applicable parts is generated. For each part in the list, the part id, part name, and the missile subsystem to which the part belongs is displayed. The generated list of parts can be carried over into the Parts Catalog subsystem, i.e., given the list of parts returned by this function, the user can enter the Parts Catalog subsystem and examine the part entries or source code, print the entries, or perform other catalog functions. Figure 28 depicts a high-level view of this function; Table 5 describes the user inputs.

The data required from the user includes information on the launch platform, target and warhead type, whether an aiding subsystem is needed, routing, seeker, the type of aerodynamic control,

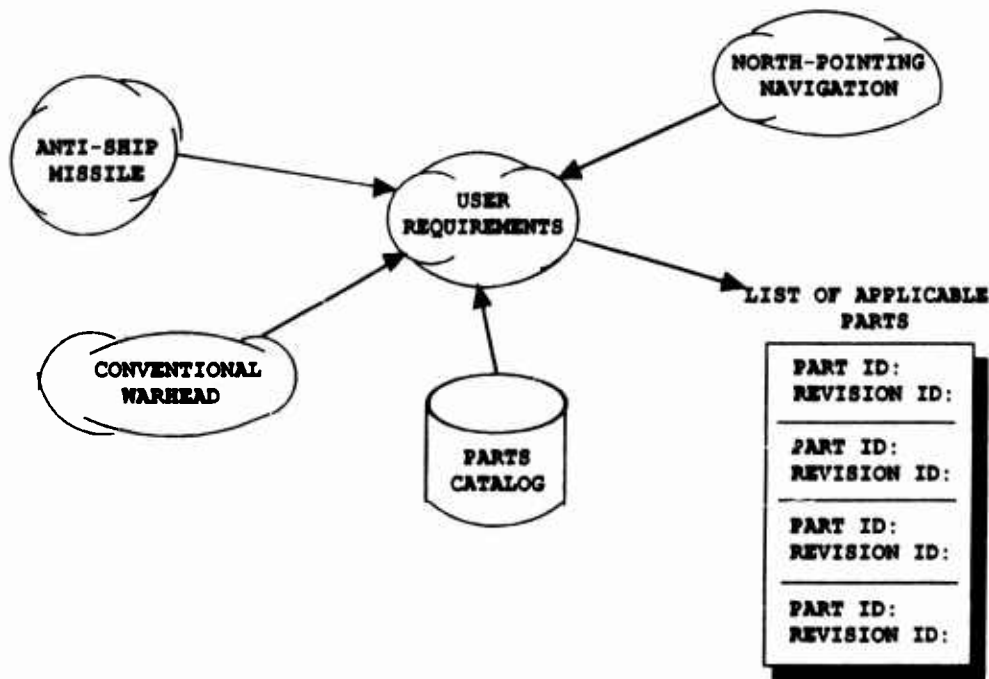


Figure 28. Application Exploration

the navigational range, and the type of interfaces. Application Exploration includes both rule-based and *parts-based* reasoning. For example, if the user indicates that the target is a ship, then the AMPEE system will conclude that a seeker is needed in the system. If a seeker is needed, the AMPEE system knows that there are no specific seeker parts, but recommends the use of math parts, the Data Bus Interface Constructor, and the Finite State Machine Constructor for construction of the required seeker software. Figure 29 shows an example of the inputs and outputs of Application Exploration.

(2) Architectural Approach

The architectural approach to parts identification is embodied in the Missile Model Walkthrough function within the AMPEE system. Missile Model Walkthrough provides the user with the ability to *walk through* a hierarchical model of missile flight software. The models used by this function are hierarchical models of missile flight software based on knowledge of the parts required for missiles of various types (e.g., it has been determined that anti-ship missiles require a particular set of software parts, and it is this set of parts that form a hierarchical model of software parts required for this type of missile). The user can traverse the model, going up, down, or sideways. The model displayed for the user shows the subsystems, functions, and CAMP parts that may be applicable for the missile described by the user. Figure 30 depicts a high-level view of this function.

TABLE 5. APPLICATION EXPLORATION — REQUIRED USER INPUTS

USER INPUTS	DESCRIPTION
Launch Type	Air Ground Surface-Sea Submerged-Sea
Warhead Type	Conventional-Submunition Conventional-Unitary Multiple-Nuclear Singular-Nuclear
Target Type	Air Fixed-Wing Air Helicopter Air Strategic-Missile Air Tactical-Conventional Air Tactical-Nuclear Air Fixed Ground Mobile Ground Surface-Sea Submerged-Sea
Range	An integer representing nautical miles. Applicable if target is any type of air.
Is aiding wanted	Applicable if target type is ground (fixed or mobile) and launch type is sea.
Aiding, seeker, or both	Applicable if target type is not mobile-ground, launch type is not air, warhead is conventional unitary, and no aiding or seeker has been specified.
Type of aiding	GPS Terrain Map Digital Scene Map Laser Radar Doppler Velocity Infrared User is queried for this information if target type is ground (fixed or mobile) or sea, or if aiding subsystem is wanted.
Seeker	Applicable if 1) User specified seeker is wanted (in query above); 2) Target type is surface-sea, warhead is not conventional unitary, and no aiding or seeker has been specified; 3) Launch type is air, target type is mobile ground, and warhead is conventional unitary; 4) Target type is any type of air and no seeker has yet been specified.
Type of seeker (non-air targets)	Imaging Infrared Radar Optical
Type of seeker (air targets)	Infrared Imaging Infrared Passive Radar Active Radar
Is ship in harbor	Applicable if target type is surface-sea.
Routing	Land Sea Applicable if target type is ground or sea.
Control Dynamics	This can be either classical or modern, but is determined by querying the user as to the required performance, robustness, and stability of his missile.
Kalman filter included	Applicable if it is determined that modern control dynamics is being used.
Error estimation	1 state 2+ states Applicable if Kalman filter is to be included in system using modern control dynamics.
Navigation near the poles	Yes No
Type of navigation	North-Pointing Wander-Azimuth Applicable if navigation is not taking place near the poles.
Interface Requirements	1553 K82 IEEE488 RS-232

Advanced Medium Range Air-to-Air Missile (AMRAAM)

Launch	: Air
Target	: Air
Range	: 25+ NM
Warhead	: Conventional
Aiding	: No
Routing	: Air
Seeker	: Active Radar
Aerodynamics	: Standard
Navigation	: Standard
Interfaces	: Data Link

Parts Selected

Navigation
Kalman Filter
Autopilot
Communications
Air Data
Coordinate V/M Algebra
Signal Processing

Data Source: Jane's Weapon Systems, 1986-1987, pp198-200

Figure 29. Application Exploration Example

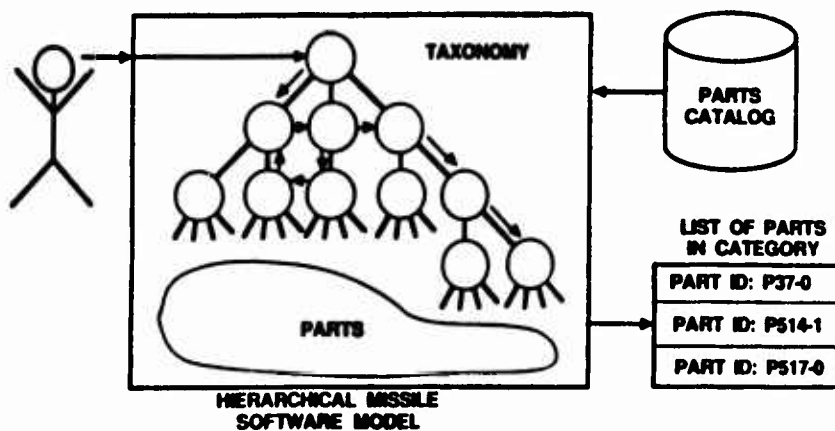


Figure 30. Missile Model Walkthrough

Missile model selection is based on the user-provided values for target type and type of warhead; CAMP work to date has indicated that launch type is not a factor in missile model selection. Additional information may be requested from the user. Once the user has provided the information requested, he will be able to view and traverse a graphical representation of the missile software structure.

The missile software systems examined during the CAMP domain analysis all contained certain subsystems regardless of the target type; additional subsystems were needed based on the target category. For instance, if the target type is some form of air target, the missile may require the following subsystems in addition to the standard ones: Kalman filter, waypoint steering, telemetry, data link, seeker. If the target category is sea and it is located in a harbor, then an aiding subsystem will be identified in addition to the usual missile software subsystems. Missiles whose targets are stationary land may require data link and aiding subsystems, while mobile land targets may also require a seeker subsystem.

The missile software hierarchy is captured within the Missile Model Walkthrough function via ART schemata and inheritance relations. Inheritance relations allow properties to be attributed to a particular class of objects and to have those properties hold true for a subclass that *inherits* from the original class of objects. A schema is used to capture the basic information about all missile software systems; this is then inherited by missiles of a particular type, such as air-to-air or air-to-sea. The particular missiles generally require software subsystems in addition to those required by the basic missile software system.

ART rules are used to check the user's input for consistency. For example, if the user indicates a target type of air and a warhead type of nuclear, a warning will be issued to the user to tell him that this is probably not allowed under the Anti-Ballistic Missile Treaty. Rules also direct portions of the user interface, controlling when the user is queried for different types of information.

(3) Testing and Operational Evaluation

Like the Parts Catalog subsystem, the Parts Identification subsystem underwent several types of informal testing and use. Development involved close collaboration between an *expert* and the subsystem developer; thus the first level of testing involved both the expert and the developer. Development and testing were iterative. The developer was able to detect and correct programming errors, while the expert was able to detect errors in the knowledge base. The subsystem underwent further user testing during the PCS training class. Few problems were detected with the system.

Although this subsystem was not heavily used during CAMP-2, it has high potential within the software development arena. One reason for its lack of use during the CAMP project, was the familiarity of the engineering staff with the CAMP parts — there was no need to use this function. Additionally, because the AMPEE system is a prototype, the Missile Model Walkthrough function took advantage of the ART Studio for the display of the model; this was sufficient to convey the information to the user, but it was less than optimal in clarity.

c. Component Constructors

The Component Construction subsystem is the third major subsystem of the AMPEE system; it comprises a number of component constructors. A component constructor is a software system that facilitates the development of application software by producing software components based on user requirements. Each constructor in the AMPEE system is based on a CAMP *meta-part*.

A *meta-part* may be either a complex Ada generic or a schematic part. A complex generic part may require data types, operators, and/or subprograms for instantiation, and may include a complex defaulting scheme; simple generic parts require only a small number of data types for instantiation. Schematic parts are parts whose design is well known, but that cannot be implemented via the Ada generic facility alone. Schematic parts consist of a *blueprint* for construction, and a set of construction rules for building a specific instance of the part. With schematic parts, there is no actual, complete, compilable piece of code until a specific instance is generated for the user. Constructors for complex generic parts assist the user in defining types, objects, and subprograms needed to instantiate the parts, and then produce the code that includes those types, objects and subprograms, as well as the instantiation(s) of the CAMP part(s). Constructors for schematic parts obtain input from the user, generate the needed code based on both the user's input and the schematic design that is incorporated into the constructor. The difference in forms is transparent to the AMPEE system user; implementation differences are discussed in Section IV.2.d. Examples of meta-parts follow.

The finite state machine is a schematic part for which a constructor has been developed. Certain types of finite state machines allow procedures to be invoked, therefore, this part cannot be captured via the Ada generic facility because procedures cannot be passed as parameters. Additionally, the variable number of states and transitions in a finite state machine are difficult to capture in generic units. All of this aside, most software engineers have a good idea of how a finite state machine can be implemented. This type of situation led to the concept of schematic parts. The Finite State Machine Constructor allows the user to specify an initial state, terminal states, state-transitions, and any actions that may be associated with either the states or transitions, and then generates the Ada code that implements this machine.

The CAMP autopilot parts are complex generic parts, and a constructor is provided to assist the user in the correct instantiation of those parts. Various data type, data object and subprogram definitions are required. The constructor has specific knowledge of the complex generic part and prompts the user for the information needed to define the types, objects, and subprograms required for the instantiations.

The Kalman Filter Constructor combines elements of both complex generic units and schematics depending on the options selected by the user. In the simplest case, the user can choose to let most of the data types information default; to the extent possible, CAMP parts will be used for types and operators. When efficiency is of concern, the user can let the constructor help him define special purpose data types (i.e., special forms of matrices) that will be incorporated into the Kalman filter code that is **generated**. These special-purpose matrices capture the active elements of a sparse matrix and unfold the operations. This eliminates the overhead involved in using full-storage matrices that operate on all elements in a matrix.

Although the CAMP component constructors generate code, they do not perform *universal* code

generation, but rather code generation in a limited domain, i.e., within the confines of the meta-part requirements. During the CAMP-1 feasibility study, it was determined that, because of efficiency requirements, universal code generation was not yet feasible in real-time embedded applications. The AMPEE system component constructors produce code that is as efficient as possible given the input supplied by the user.

The constructors reduce the user's need for both detailed Ada knowledge (because the code itself is generated for the user) and for detailed knowledge about the software parts on which the constructors are based. A straightforward user-interface is provided to facilitate requirements specification by the user. These requirements are analyzed by the constructor and tailored Ada code is produced. The constructors are intended for use by application developers, and can be used both for trying out *what ifs* and for actual software development.

The user must have some knowledge of Ada because he will often be prompted to provide information needed to define data types, data objects, and subprograms. He must also have some familiarity with the application area in order to be able to produce meaningful output. The AMPEE system parts catalog can be used to obtain detailed information on the parts on which the constructors are based, thus the user will not need to be intimately familiar with the parts themselves.

The constructors provide the user with the ability to generate tailored Ada components, modify the component requirements (either in place or after making a copy) and regenerate the component, and delete the requirements upon which the components are based. Component regeneration may be necessary if, after a user has generated a component, he realizes he must alter some of the requirements.

Although each meta-part is associated with its own component constructor which guides the software generation process, and each constructor has its own requirements and design, the top-level design of all of the constructors follows the basic paradigm of inputs-processing-outputs (see Section IV.2.d). The inputs are the component requirements provided by the user, and the major output is the tailored Ada software component. Processing consists of interacting with the user, checking of input data, internal processing to transform the data into a usable form, and writing out the application-specific Ada code.

The constructors that comprise the AMPEE system are summarized below.

- **Kalman Filter Constructor**: The Kalman Filter Constructor provides the user with a tailored version of the CAMP Kalman filter parts, plus the data types that are needed to support Kalman filter operations.
- **Finite State Machine**: The Finite State Machine Constructor will construct one of three varieties of finite state machines (Mealy machine, Moore machine, or a finite state machine with no actions).
- **Pitch Autopilot Constructor**: The Pitch Autopilot Constructor provides an Ada pitch autopilot component, plus the required data types, filters, and a limiter.
- **Lateral/Directional Autopilot Constructor**: The Lateral/Directional Autopilot Constructor provides an Ada implementation of a lateral/directional autopilot, plus the required data types, filters, and limiters.
- **Navigation Subsystem Constructor**: The Navigation Subsystem Constructor provides a single navigation subsystem composed of selected navigation computations.
- **Navigation Component Constructor**: The Navigation Component Constructor provides a set of individual navigation computation components.
- **Data Bus Interface Constructor**: The Data Bus Interface Constructor provides the user with a general-purpose interface to a data bus.
- **Data Type Constructor**: This constructor assists the user in the definition of various Ada data types.
- **Abstract Processes Constructors**: There are four constructors in this category: a Task Shell Constructor, Time- and Event-Driven Sequencer Constructors, and a Process Controller Constructor.

(1) Design Paradigms

A standard paradigm for constructor design and a methodology for component constructor development was developed under CAMP-2. The standard design paradigm promotes consistency, ease of integration, and standardization of user interfaces. The standard methodology facilitates the development process; it stresses many informal reviews as work progresses, and is iterative in nature. The paradigm for constructor design covers three areas:

- The user interface
- The processing or analysis phase
- The code generation or synthesis phase

The need for constructors can be identified either by projects who recognize the need for a constructor or through domain analysis performed by a *parts* group. The constructors that comprise the AMPEE system were identified during the domain analysis of CAMP-1. Constructors can be developed both to assist in the use and tailoring of complex Ada generic parts, and to produce tailored Ada code

from schematic designs. Once it is thought that a new constructor is needed, an intensive analysis should be performed to determine if there is sufficient demand for such a constructor to warrant the non-trivial development cost. For example, the Kalman Filter Constructor comprises some 8000+ lines of Lisp/ART code and has access to another 2700 lines of code in common utilities.

The user interface front-end elicits requirements from the user for the software component that is to be generated; the processing portion then converts the requirements into an internal representation; and the back-end, or synthesis phase, generates the required Ada code for the user. Figure 31 illustrates this design paradigm.

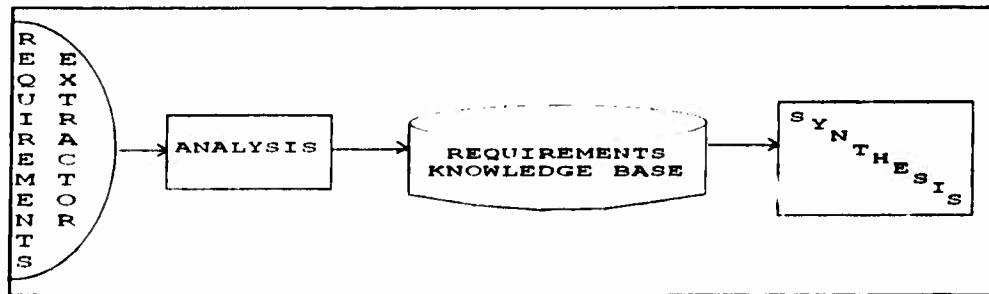


Figure 31. Constructor Design Paradigm

The user's requirements undergo analysis for completeness and consistency, and are then stored in an intermediate form. The synthesis phase generates the Ada code from the requirements provided by the user. The code can consist of generated Ada components, instantiations of complex Ada generic units, or a combination of instantiations and generated code.

Early in the development process, the constructor developer formulates preliminary requirements and questions for the *expert*, i.e., the Ada parts developer. The parts developer must then consider these requirements with the following issues in mind:

- What CAMP parts can or should be used?
- What alternatives should the user be presented with (e.g., Should he only be able to put together CAMP parts when constructing the component or should he also be able to provide his own parts?)?
- What information should be elicited from the user (i.e., the wording of the interface is important — it should be in the domain language)?
- What are the implications of different choices made by the user?

Following this preliminary work, the constructor developer and the parts team member responsible for the design of the corresponding Ada part should meet to delineate the scope of the constructor, clarify requirements (inputs, processing, and outputs), and determine acceptance criteria. After this initial meeting, the constructor developer begins defining the dialog that will be conducted with the user. This process includes the development of preliminary *screen flow* diagrams. These diagrams depict the actual user interface for the constructor. Early development of these diagrams helps to point out omissions in the requirements and misunderstandings between the intent of the Ada part designer and the constructor designer.

After several iterations on the preliminary screen flow diagram, the constructor developer produces a complete screen flow diagram that depicts the user interface for the four constructor functions — *Generate*, *Browse/Modify*, *Copy/Modify*, and *Delete*. A standard symbology for the screen flow diagrams has been developed (see Figure 32).

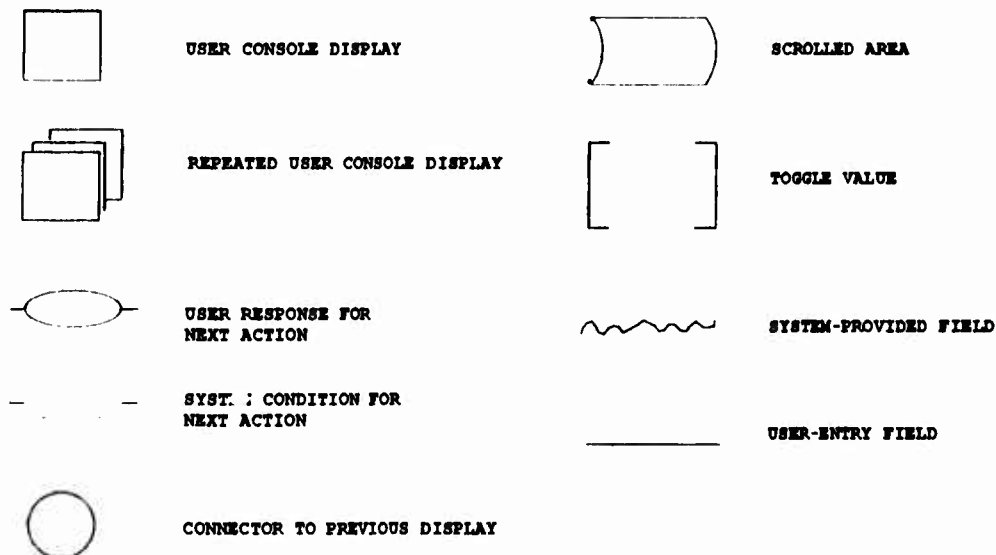


Figure 32. Screen Flow Symbology

A second diagram depicting the high-level view of the constructor is also produced. This diagram depicts the CAMP parts that will be used, the packages that will be provided by the user, and the packages that will be output from this constructor. It shows the major options available to the user. As an example, Figure 33 depicts the top-level view of the Kalman Filter Constructor. The user has several choices to make. He can choose between using Compact or Complicated H parts. He also has a choice in the provision of required data types, operators, and subprograms: he can allow the data types to default to those provided by the constructor, he can provide his own package, or he can define his own data types interactively. The output from this constructor consists of a data types package and the actual Kalman filter component.

Once these two diagrams are completed, they should be reviewed by a team that consists of the Ada parts designer, the constructor developer, and the chief Ada designer. The constructor design generally encompasses several options for construction of the component (these are clearly identified in

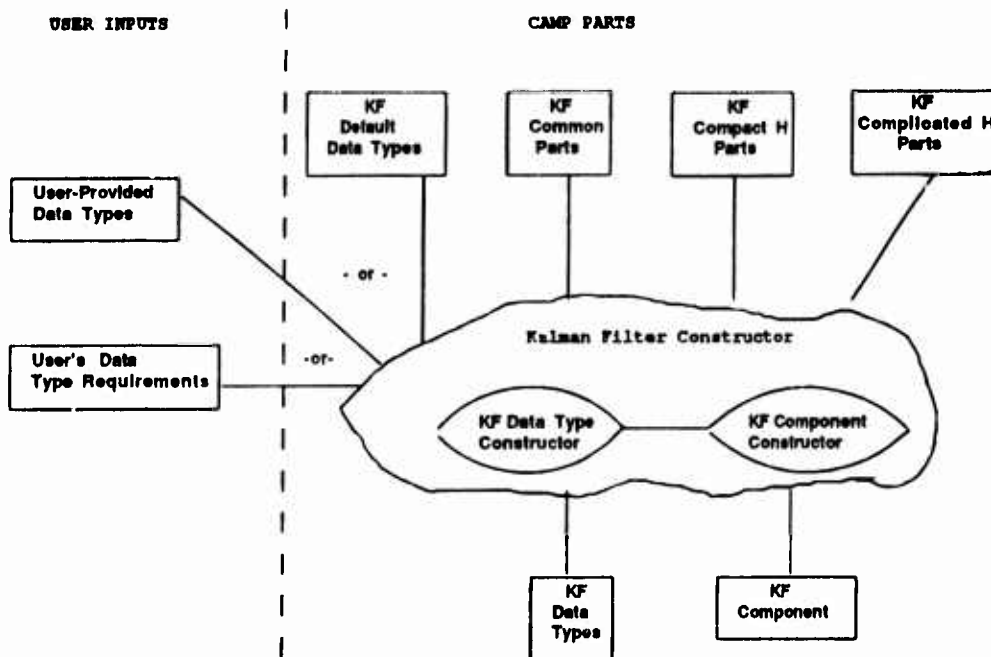


Figure 33. Kalman Filter Constructor — High-Level View

the screen flow diagram). During the review meetings, a determination is made of which options will be implemented first, and the priority of the other options. If it is determined that the screen flow meets the requirements of the constructor, then the PCS developer may proceed with further design and implementation of the component constructor. This process of diagram development and review may be iterative.

Upon successful completion of the diagrams, the constructor developer should produce a program design language (PDL) description of the constructor and the data structures required for implementation. These may be reviewed informally prior to implementation. Once the design is complete, the constructor developer can begin implementation, concentrating on the options that were assigned the highest priority.

The entire design and development process is iterative in nature. The approach developed during CAMP-2 emphasizes many informal reviews along the way. This serves several purposes: it ensures that the constructor developer is *on track* with the requirements and design of the constructor, and it facilitates communication with the parts team (i.e., they are kept informed of activities within the component construction team).

(2) Constructor Implementation

A standard structure has been developed for implementing constructors; this not only facilitates integration but also reduces the development time for new constructors. Common routines have been developed for AMPEE system entry and exit, and common constructor functions. Utilities consist of routines that perform error checking, and low-level processing. User-interface functions provide common routines that handle different types of menus and forms that are widely used. The AMPEE system user interface utilizes host facilities known as *presentation types* to control the type of input that the user may provide. This facility is an extension to Common Lisp, the language used for AMPEE system implementation. The *presentation type* facility allows data types and error checking routines to be defined to limit the range of valid inputs.

The constructors are controlled by a *constructor executive* that is constructor-dependent. The *executive* is responsible for *evaluating* the functions in the *function list* that consists of the Lisp functions that must be executed for the particular constructor function (i.e., *Generate*, *Browse and Modify*, *Copy and Modify*, and *Delete*). A global variable is used to keep track of the current location in the function list.

(a) Types of Constructors

Although all of the constructors follow the same basic design paradigm, there are implementation differences between constructors for complex generic units and schematics; these are transparent to the end-user. Constructors for complex generic parts encode knowledge about instantiation of those generic parts; this includes information on the data types, operators, and subprograms that are needed for instantiation. Constructors for schematic parts encode a *blueprint* or *schematic* of the component that is to be generated; knowledge about Ada coding procedures and efficiency issues is also encoded in the constructors, e.g., within the Finite State Machine Constructor a decision is made on whether to use a *case* statement or an *if-then-else* based on the number of options that will be processed.

(b) Code Generation

Once all of the information needed to generate a component has been obtained from the user, code generation begins. The data is extracted from the requirements schemata, and the appropriate generic units are instantiated, or needed code is generated based on an existing blueprint.

The code generation phase requires no further interaction with the user. It is driven by requirements provided by the user and encoded in the constructor itself. The code generation process is unique for each constructor; the complexity varies considerably among constructors. In general, the data type definitions are generated first, followed by instantiations of CAMP parts and/or production of new code.

(3) Testing and Operational Evaluation

The individual constructors and the entire Component Construction subsystem underwent various levels of testing, as did the other subsystems that comprise the AMPEE system. Constructor testing consisted of the following:

- Interface and operational testing by the developer
- Informal user testing performed by the PCS development team
- Code inspection performed by a combination of the PCS team and the parts team
- Compilation testing which was generally performed by the constructor developer
- User testing by the the PCS training class

Several of the constructors were subjected to further testing and use, including formal testing via the CAMP Ada parts test procedures, and use by the 11th Missile team. When formal testing was performed on the output from a constructor, it was conducted by a member of the CAMP parts team.

Overall, users found the constructors to be a useful concept. The major drawback is that their implementation is closely linked to the meta-parts they represent, therefore, changes to the meta-parts generally necessitate changes to the constructors. This is an implementation problem, not a problem with the concept of component constructors. Additionally, the constructors require more domain specific knowledge to run than the Parts Catalog subsystem, but that is to be expected.

2. PCS IMPLEMENTATION

The AMPEE system was originally conceived as an expert system, but it was found that for the most part an expert system was not required. This divergence from the original concept resulted from a combination of factors, including one that is quite common. In reviewing the literature, it is evident that as a problem becomes better understood, a sequential solution is often found to a problem that was originally thought to be non-deterministic.

a. System Architecture

The AMPEE system is implemented using ART (the Automated Reasoning Tool from Inference, Corp.), a commercially available expert system shell, and Common Lisp. It is hosted on a Symbolics 3620 computer, a single-user Lisp workstation, and takes advantage of Symbolics extensions to Common Lisp. Figure 34 depicts this architecture.

A Lisp machine differs from a conventional workstation such as the DEC MicroVAX, in that its architecture has been developed to support the Lisp programming language (although it can be used for other languages as well), i.e., it is intended more for symbolic computation than arithmetic processing. Symbolics provides an extensive integrated development environment. This includes on-line help, documentation, and debugging facilities, and incremental compilation of functions in the editor. Extensive user interface facilities are also provided in the form of extensions to Common Lisp.

An expert system shell is a software system that provides a means for capturing knowledge, and

an inferencing mechanism to work on that knowledge; the application developer provides application-specific knowledge in the form of facts and rules.

The AMPEE system was originally hosted on a DEC MicroVAX and utilized DEC Common Lisp and *beta* versions of VAX-compatible ART. The advantages and disadvantages of each implementation will be discussed in the following paragraphs.

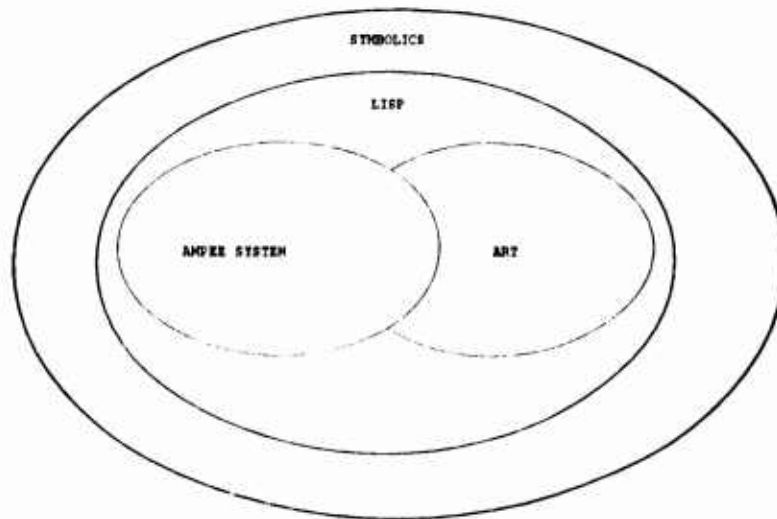


Figure 34. AMPEE System Architecture

(1) Hardware

The CAMP-1 PCS feasibility study included an evaluation of an off-the-shelf expert system tool for use in the PCS. CAMP-1 also required the use of a widely available processor both in the evaluation of the expert system tool and the PCS feasibility study, thus the DEC VAX family of computers was selected.

The VAX implementation of AMPEE, which began as a proof-of-concept implementation under CAMP-1 and continued into the prototype stage under CAMP-2, made use of DEC Common Lisp, DEC Forms Management System (FMS) for the user interface, and ART for the knowledge structuring. ART was selected in light of the hardware selection (at the time of the CAMP-1 contract — September 84-85 — Inference appeared closest to producing a full-scale, production quality, expert system development tool for the VAX).

DEC FMS was used for the interface in part because the beta versions of ART did not provide adequate tools for the development of a full-screen user interface. Although FMS provided a significant improvement over developing an interface from scratch, it required a considerable initial effort to use. The CAMP PCS team developed utilities for use with the FMS forms; these handled things such as forward and backward scrolling, cursor positioning, etc. The Lisp-compatible version of FMS did not provide automatic type checking of user-provided data, thus, all such processing had to be performed in the application code. Additionally, a graphic interface was not possible.

In comparison to other VAX languages, VAX Lisp was slow and consumed large amounts of space. Additionally, a production quality version of ART was never available during the time AMPEE was hosted on the VAX; expected delivery dates slipped continually, and the versions that were available were not error-free. The problems experienced with ART drove the AMPEE implementation deeper into Lisp.

The main advantage of the VAX implementation of AMPEE was that it was hosted on widely available hardware, and thus could be used by a larger audience. FMS was relatively inexpensive, and hence was not a deterrent to using the AMPEE system.

The application was ported to a Symbolics during CAMP-2. A major factor in the move from the VAX platform to the Symbolics was that the VAX version of ART, which had been implemented in VAX Common Lisp, was being re-hosted in C. Inference Corp. intended to provide a subset of Common Lisp that could be called from ART, but, at least initially, the full Lisp functionality that the PCS development team had come to rely on would not be available. Thus, a decision had to be made to either port to the C-based version of ART on the VAX, or port to a Lisp machine. Porting to the C-based version would require rewriting significant quantities of Lisp code and reworking parts of the application. Porting to a Lisp machine would require redevelopment of the user interface, but presumably a complete implementation of ART and Lisp would be available.

The port from the VAX to the Symbolics required not only a complete re-implementation of the user interface, but also a change in the type of interface. On the VAX, the forms and menus were, for the most part, full screen, whereas on the user interface developed for the Symbolics version consisted of pop-up menus and forms that generally filled only a portion of the screen.

The Symbolics provided a good development environment. Rapid development of prototype interfaces was possible, although more work was required for development of *custom* interfaces.

(2) Software

Both ART and Lisp were used in the implementation of the AMPEE system. ART is a programming language that bears some resemblance to Lisp, although its functionality is quite different. As mentioned previously, ART is an expert system shell intended for both rapid prototyping and production of expert systems. It provides a means of capturing knowledge in the form of rules and schemata, and a means of invoking, or firing, those rules. Although the basics of ART are fairly simple to master, it is a complex tool. Before utilizing such a tool, it would be beneficial to determine which of its features are likely to be needed, and determine if some or all of the needed features are available in a simpler and more portable package or language.

The AMPEE system makes only limited use of ART functionality. It is used throughout the AMPEE system for data structuring (via the ART schema system), and within the Parts Identification subsystem for consistency checking and interface control (via a small number of simple forward-chaining rules), and for display of the missile software hierarchy within the Missile Model Walkthrough function. ART provides many more features that are not used in the AMPEE system, such as backward chaining rules and the ability to explore alternative scenarios via the *viewpoint* mechanism.

The use of ART for system development imposes a number of limitations on deployment of the final system:

- Portability: Although it is available on a fairly wide range of processors, its use does cut down on the portability of the application.
- Cost: There has been a general downward trend in the cost of high-end expert systems, but, the cost of such a tool can be prohibitive to some potential users. Some vendors, including Inference, also market a run-time system separately from the development environment.
- Compatibility: ART must stay current with the operating system under which the user's machine is running.

The PCS team developed a significant quantity of reusable software that was used throughout the AMPEE system. This benefitted not only the developers but also the end-user. Much of the reused code was for user interface functions, thus the user was presented with a more uniform interface than might otherwise have been possible.

(3) User Interface

There are several basic types of data entry/display used in the AMPEE system interface. They are explained below.

- Single-choice menu: The user mouses on his selection. Most of these menus include *Pop* as a selection; this allows the user to backup to previous screens to examine or re-enter data.
- Multiple-choice menu: These are two part menus, whereby the user selects as many of the items in the choice portion of the menu as he would like, and then selects the action option from an embedded menu. The action options are *Do It* (take the options selected by the user), *None* (no options desired), *All* (select all of the options), and *Quit* (exit this screen).
- Fill-in-the-blank field: The data type of the response, or a default value is generally displayed in the field. The user mouses on the response area for the field, enters a value, and presses the *return* key to proceed to the next field. If the user is unable to mouse on a field, it is not changeable.
- Multiple-choice field: All of the options are displayed to the user; he can make or change a selection by mousing on item. Default values are displayed in bold-face type.

- Display-only form: These are used to provide instructions, or to display information to the user that he is unable to alter.
- Scrollable list/menu: This is a list of data that may be more than one windowful in length. It may be scrolled by clicking left or right after the scroll arrows are visible. The scroll arrows can be made visible by knocking the mouse arrow into the left margin. For menus of this type, the *next action* choices (i.e., *Done*, *Pop*) generally appear as menu choices.
- Scrollable form: This is a form that is used to display a list of data items or text that may be more than one windowful in length. The user may scroll either by using the scroll key or the mouse. At the bottom of the form are two option boxes, *Done* and *Pop*. To exit from this type of form, the user must mouse on one of these two boxes. If *more* mode is in use, the user must scroll past all of the *more* prompts before his choice of *Done* or *Pop* will be processed.
- List-input form: This type of form allows the user to provide one or more values of a particular type. The user is prompted for only one item at a time. Data is entered by mousing on the prompt, supplying a value, and pressing the *return* or *end* key. Another prompt line will appear. To terminate processing the user can either mouse on the end option at the bottom of the form, or press the *end* key. In general, a user can *pop* from this type of form by entering *pop* on the last prompt line. To delete entries, the general procedure requires the user to mouse on the entry to be deleted, and then enter *nil*. Specifics may vary somewhat from form to form.

b. Parts Catalog

The basic goals of a software parts catalog are to facilitate reuse of pre-built software parts, facilitate configuration control, and provide a foundation for a parts composition system. It is basically a data base application, although the AMPEE Parts Catalog has been implemented using ART schemata and Lisp in order to provide the user with a single integrated parts composition system.

ART schemata are used to capture the actual catalog data. There is one schema for each catalog entry, and one schema *slot* for each catalog entry attribute. A *slot* is comprised of a slot name and a slot value, thus, the slot name represents the catalog attribute (e.g., date-of-last-change-of-entry), and the slot value represents the attribute value for a particular catalog entry (e.g., 12-02-87).

Some of the catalog entries are *textual*, therefore, their value is not actually stored in the catalog schema. Textual attributes are used to capture attribute values that are of indeterminate length. Their actual value is stored in a file, and the name of the file is stored in the schema slot associated with that particular attribute. When a user wants to view a textual attribute, the contents of the file are fetched and displayed. When textual attributes are added or modified, the AMPEE system user is put into an editor. Deletes of textual attributes do not physically take place until the user exits the AMPEE system and confirms that he wants the catalog changes to be saved.

The remainder of the Parts Catalog subsystem is implemented using Lisp. The ART data structures are accessed via Inference-provided Lisp functions. This subsystem also makes use of an editor

for the entry of information for textual attributes. Additionally, the print function makes use of a commercially available text processing program to format the catalog entries.

There are several deficiencies in the current implementation. Among them is the fact that there can be as many as fourteen textual attributes provided per catalog entry. Access to textual attributes is relatively slow compared to access to other attributes because the request must go through the file system. The file system can also become cluttered with attribute files, particularly if there is a minimum size imposed on all files by the operating system. The textual attribute files also contribute significantly to the overall time needed to exit from the AMPEE system. If the user chooses to not save the catalog changes from the current session, the AMPEE system exit routines examine the attribute files, deleting any that were created during the current session.

Another problem with the current implementation is the amount of time needed to load the catalog initially, and to save it after completing a session. Both times are directly tied to the number of catalog entries. Perhaps a re-examination of what is cataloged is in order.

c. Parts Identification

The Parts Identification subsystem is the only portion of the AMPEE to use ART for anything more than data structuring. It makes use of both simple forward chaining rules and facilities within the ART Studio. The ART Studio contains several functions for *browsing* the current state of an ART knowledge base, including one that permits viewing of the *inheritance network*; it is this function that is used for the display and traversal of the missile models in the Missile Model Walkthrough function. Although this function serves its purpose in a prototype implementation, clarity of the display for this application is less than optimal.

d. Component Constructors

Each constructor within the Component Constructor subsystem incorporates one or more ART schemata that capture the requirements needed to generate a specific component for a user. These schemata are referred to as *requirements sets*; they are accessed by Inference-provided Lisp functions that are called from within the constructor. Storage of the requirements sets allows them to be recalled at some future session, and either modified or used as is to generate additional components.

3. FUTURE DIRECTIONS

The prototype PCS developed under CAMP has proven that tools can be developed to facilitate the use of reusable software. Development of this prototype has pointed out both potential problem areas within the current implementation, and areas for further development and investigation. Listed below are some areas for further work/investigation.

- Parts Catalog

- Restructure the parts catalog. Currently, almost anything can be cataloged; this has not proven to be necessary, or particularly useful. It can result in confusion on the part of the

user when he is confronted with 1100+ catalog entries when there are only about 450 parts. This discrepancy is the result of cataloging specifications and bodies separately, and of also cataloging encapsulating packages which are not classified as parts by the definition developed during CAMP-2. It is also possible to catalog generic formal parts and context clauses, in addition to generic and non-generic package, task, and subprogram specifications and bodies.

Restructuring the catalog could benefit the Parts Catalog in another area -- within the Examine Part function. This function allows the user to examine the source code for the cataloged entity. If the entity is a part that is encapsulated within a package, the user is presented with the entire file that contains the particular entity of interest (the user does have the option of having the header comments stripped out of the file before it is displayed). This can be both an annoyance and a deterrent to use. It is inconvenient to step through a large file to find a deeply embedded entity. Additionally, it can be confusing, and cause the user to think that the system is not operating correctly.

- It might be beneficial to provide the user with functions that act more directly on the part hierarchy. Currently, the user can obtain information on the hierarchical structure of the parts via an examination of the *built from* and *used to build* attributes. There should be a more straightforward way to obtain this information (perhaps graphically).

- Parts Identification

- Extract the essence of the Parts Identification functions so that the basic mechanism can be applied to domains other than those covered by the CAMP work.
- Expand the missile models to permit finer granularity in the identification of parts for the user.
- Expand the knowledge base to incorporate more domain knowledge, the goal being the identification of parts that can be used to build needed parts.
- Smooth the transitions between PCS functions, and provide greater *carry-over* between them.

- Component Constructors

- The concept of component constructors is a valuable one, but the approach to implementation of constructors is an area that could benefit from further work. The approach used in the AMPEE system ties the constructors (for complex generic parts) intimately to parts that they utilize. This can be a problem if the part(s) on which the constructor is based change in areas that are relevant to the production of code by the constructor.

An alternative that bears further exploration is the concept of a *constructor constructor*, i.e., a generalized software constructor that would generate specific constructors. One way to do

this would be by embedding commands within the reusable parts themselves that would indicate the information that would be required from the user in order to generate the tailored Ada components that are needed. The parts could then be run through a preprocessor to produce the appropriate user queries. Code generators and facilities to permit data type definition or provision outside of the constructor would also be required. In essence, this would be a smarter constructor, where less of the information is hard-coded in the constructor itself.

- General

- The entire AMPEE system, as it is currently implemented, is not very portable. Although it has been implemented using ART and Lisp, the majority of the interface is implemented using Symbolics extensions to Common Lisp. Additionally, the use of ART limits the potential users to those who have a compatible version of ART available to them.
- It is a time-consuming process to load all of the files associated with the AMPEE system, but certain things can be done to alleviate this problem. Provided the user has sufficient disk space on his machine, he can load all of the AMPEE system files, and create a file that captures the current machine environment; it is then possible to avoid loading all but the user-changeable AMPEE system files each time the user wants to run the AMPEE system. The user-changeable files include the catalog itself, and the collection of requirements sets created by the user. Both of these files can be updated by the user, therefore, it is not recommended that they be made a part of the environment that is saved (i.e., they should be loaded each time the user starts a new session. The extent of the inconvenience of doing this is somewhat dependent upon the amount of memory that the user has on his machine. In the AMPEE system development environment, loading the catalog took in excess of fifteen minutes (note that load time is also dependent upon the number of catalog entries).
- Response time is another potential problem area. Users have been conditioned to expect a fairly rapid response when interacting with a computer. Some of the items that factor into AMPEE system response time are the location of the files that are accessed (i.e., is there activity across the network or are all of the required files resident on the host machine), the number of catalog entries (for various AMPEE Parts Catalog functions), whether or not the underlying Lisp code has been compiled or not, and the amount of memory on the host machine. The AMPEE system, as it is currently implemented, can leave the user *hanging* while various user requests are fulfilled.
- Another area that could be improved to enhance usability, is the interconnectivity of AMPEE system functions. Although there is some carry-over between functions, it is limited, e.g., there is carry-over from the catalog search function to other catalog functions that operate on existing catalog entries.

SECTION V

THE ADA LANGUAGE AND SOFTWARE REUSABILITY

Support for reusability has been a key goal of the Ada language since its development began in the 1970's. Reusability includes the ability to transport code between different machines and the ability to transport code between applications. Standardization on a single language specification and prohibition of modifications to create subsets or supersets of the language have largely achieved the first component. Complete Ada applications have been transported between widely disparate machines, with minimal changes to the source code. The success of this type of reusability is, of course, limited by machine dependencies of the code and the type of application involved.

For reusability to truly succeed, Ada code must be transportable not only between machines, but also between applications. While there have been successful cases of this type of reusability, the CAMP project has concluded that portions of the current Ada standard inhibit the ability to transport and reuse code between applications. These conclusions are based on problems uncovered during the implementation of the CAMP parts and the 11th Missile Application; therefore, the problems are seen as valid issues which must be addressed, and not as conjectural speculation on potential use of the language or as the result of a specific effort to find fault with the language.

Ada language issues raised during CAMP include language definition and language restrictions. In some cases, the standard leaves to the compiler implementor key decisions which can affect the ability of a compiler to handle code developed for reuse. Furthermore, as discussed in Section VII, the Ada validation capability does not adequately test all of the standard Ada features which are required for implementation of reusable software. Additionally, the standard lacks certain specific features which could further enhance reusability, especially for the design of special interfaces.

This section of the report discusses areas where Ada's support for reusability of code between applications can be improved; examples from CAMP implementations will illustrate the problems. Recommendations are also made for implementing these improvements as a part of the Ada 9X revision process.

I. SEPARATE COMPILATION AND GENERIC UNITS

Ada generic units form the key constructs of reusable software. Sections II and VI of this report discuss the use of generic units in CAMP parts development and in the CAMP development methodology.

Part of the CAMP development methodology includes the separate compilation of generic subunits. This approach facilitates development and maintenance by reducing the size of compilations and the requirements for recompilation should subunits change. This approach is also consistent with the goal of Ada to support modularity.

The development of reusable software based on generic units is impeded by ambiguity in the Ada language standard. Support for separate compilation of generic units is not a required Ada feature; therefore, it is not addressed in the Ada Compiler Validation Capability (ACVC) tests. The exact statement on this issue occurs in Section 10.3, Paragraph 9 of the Ada Reference Manual:

"An implementation may require that a generic declaration and the corresponding proper body be part of the same compilation, whether the generic unit is itself separately compiled or is local to another compilation unit. An implementation may also require that subunits of a generic unit be part of the same compilation." (Reference 8)

This statement is somewhat ambiguous. An implementor could provide a compilation system which would require that an entire package be placed in one compilation. Since on most systems a compilation would correspond to a file, the file would contain the main unit and unit body plus all subunits, where the unit is a generic package or subprogram, or any combination of nested packages or subprograms within the main unit. Any change anywhere within the unit would require recompilation of the entire unit and body.

The effect of this requirement would be disastrous for the developers of a reusable software library, such as the CAMP parts library. Figure 35 illustrates potential compilation structures for a simple package encapsulating two generic units. (Note: This is an extremely simple case; generally, library packages will be far more complex. Also, the compilation system assumed here will perform recompilations, whether or not units have changed. More sophisticated systems could permit incremental compilation, which would only affect units which actually require recompilation.)

- Compilation 1 is the most desirable structure, allowing specifications, bodies, and separate units to be physically located in separate source code files. This structure supports ease of development and maintainability through separate compilation of specifications and bodies.
- Compilation 2 allows the specification to be located in a separate file, but requires all bodies to be located in the same source code file. This structure increases compilation and maintenance time, but has no affect on package use.
- Compilation 3 has the same result as Compilation 2. This structure would be required by a system that did not permit separate subunits of a generic unit.
- Compilation 4 requires a specification and all corresponding bodies to be located in the same source code file. This structure requires complete recompilation of the specification and body whenever any part of B or C changes. In addition, because units which import A are rendered obsolete by the recompilation of A, the importing units have to be recompiled, as well.
- Compilation 5 does not make use of packaging and, therefore, each of the units can be located in separate source code files. This obviates the need for extensive recompilation. Only B or C need to be recompiled if they change, and any units importing the changed unit also require recompilation. However, the library would become unmanageable because of the number of units.

For the CAMP library, the requirement to structure compilations according to Compilation 4 would typically mean recompilation of from 500 to 2000 lines of code, depending on the specific unit, any time

COMPILATION

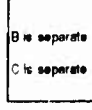
FILE STRUCTURE

1 .

A (spec)

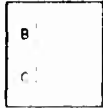


A (body)

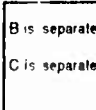


2 .

A (spec)

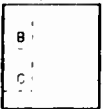


A (body)

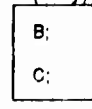


3 .

A (spec)



A (body)



4 .

A (spec)



5 .



= a single file

Figure 35. Ada Generic Compilations

an encapsulated unit was changed. If specifications and bodies must be in a single compilation, the entire CAMP software library could potentially be affected as a result of a single change in the body of a basic unit. Compilation 5 would split up large units into small constituents; however, this would require massive context clauses and would enlarge the already complex CAMP configuration management task.

For users of a reusable Ada software library, the compilation restriction would be equally serious. If the specification to a library package changes for any reason, all software dependent on that package, would have to be recompiled. It is hoped, however, that changes to a specification in a reusable library would be extremely rare. However, if an Ada compilation system required that the specification and body of a generic unit be in a single compilation, as in Compilation 4, a change to a single statement in the body would require recompilation of the specification as well. The resulting recompilations could ripple through the user's system and require extensive, if not complete, recompilation of the user's software.

The ambiguity of the standard in this area also permits implementors to exaggerate their claims about their compiler. Because no standard on separate compilation of generic units exists, an implementor can claim support for the feature, yet may have almost no support beyond separating specification and body into two separate compilations. Attempts to separately compile units beyond this level can result in any number of compiler errors, as reported in Section VII. The fact that nearly every implementor does claim support for this feature indicates that it is a feature desired by most compiler users.

The CAMP experience establishes the need for making separate compilation of generic units a required feature of Ada. In order to provide continued support for software reuse, the 9X revision must consider this need. Numerous tests of support for the facility were developed during the CAMP-2 project; these tests can be used to measure the claims of compiler implementors in the interim. These tests form part of the CAMP Armonics Benchmarks described in Volume III of this report.

2. OPTIMIZATION

The success of Ada requires the availability of optimizing compilers. Without significant optimization, Ada will never achieve the throughput or memory restrictions imposed by requirements for most embedded software systems. Additional optimization to address generic units will also be needed because of the heavy use made of generic units in reusable software.

Of particular concern to developers of optimizing compilers is the issue of optimization across unit boundaries. Several optimization issues must be addressed:

- How will an individual user utilize the objects in a package?
- Will all users of the same package require the same optimizations?
- Where another unit imports objects from the original, what effect will use of the objects of the original unit have on optimization of the user's objects?

With generic units these requirements become even more difficult to address. By design, the generic unit must meet the needs of numerous end-users. These needs must be tailored by data structures and specific sets of operations on the data. While the CAMP parts have been designed to address many of the typical efficiency needs of embedded systems to limit or even eliminate inefficiency due to reusability, the success of reusability itself depends on compilers optimizing the final code once the generic unit is instantiated.

Another form of optimization is the generation of code based on the actual parameters in a generic

instantiation. Presently, the actual parameters do not affect the code generation; compilers either generate new code for each instantiation, even if actual parameters are the same, or they generate only one body regardless of the number of instantiations. In the former case, size can grow even though each instance is a mere copy. In the latter, the compiler must generate additional code to handle specific calls to each instance. The additional code will be generated even if there is, in fact, only a single instantiation.

Figure 36 shows these two methods plus a hybrid method that overcomes the deficiencies of the other two. This *different body* method will perform three types of instantiations:

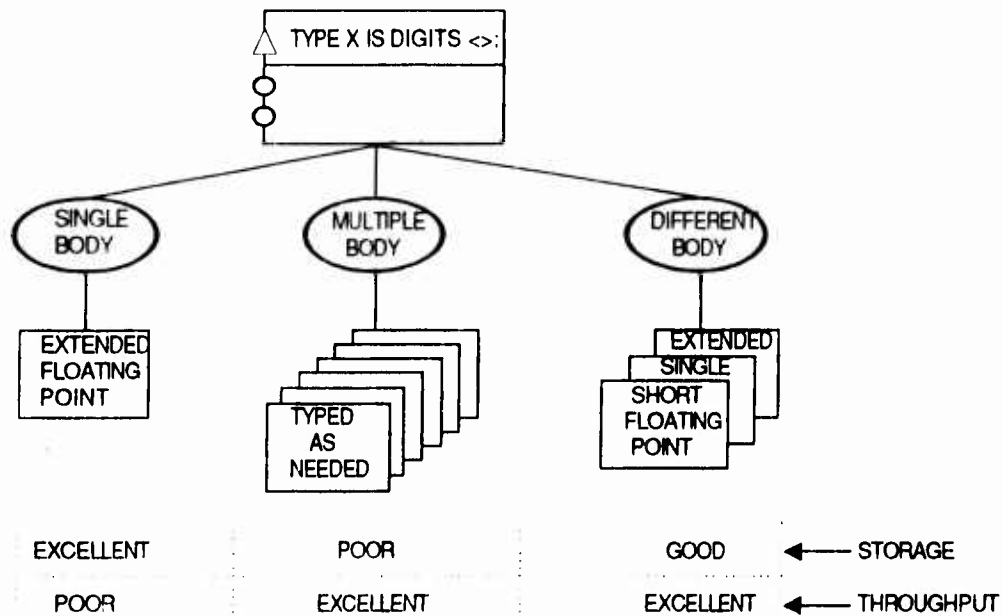


Figure 36. Methods of Generic Instantiation

- Generate multiple bodies for truly distinct instances
- Generate a single body, with no additional case-specific code, if there is only a single instantiation
- Generate a single body for multiple instantiations using the same actual parameter types

This approach assumes global optimization across packages. Because it is not generally possible to know in advance how a potential user would use the generic unit, the optimization must occur following all instantiations.

The CAMP parts library provides compiler developers with an excellent set of examples for dealing with optimization issues. Within the parts structure itself, there is extensive reuse of objects between parts. Optimization addressing the needs of the parts themselves could address many concerns that arise in the development of a library of parts. The 11th Missile Application demonstrates the use of parts in building an application, and can provide guidance to compiler developers in meeting the needs of the end-user of the parts.

3. TASK PRIORITIES

Task priorities are used to indicate the relative priority of one task over another in the allocation of system resources. In general, those things that have to be done quickly or on a precise time schedule are given the highest priority. For example, within the 11th Missile Application, the interrupt handlers which process messages to and from the 1553B bus are given the highest priority in the system because the interrupts can come as often as every few hundred microseconds, and each must be handled before the next one comes in.

Tasks with low priority generally occur relatively infrequently and do not have tight timing constraints. For example, within the 11th Missile Application, the ISA.Monitor queries the ISA status every minute. If the actual delay between queries is 62 seconds instead of 60, no major problems will result. Similarly, the Status_Generator runs twice a second to generate an operator status display. If one or more runs get skipped, there is no adverse effect on the system. Neither of these tasks can be allowed to delay the completion of the higher priority tasks.

Currently, the Ada language standard requires that task priority be established by a *static* value, thus, not only must the priority be fixed, it must be fixed by a simple constant expression. Reusability could be enhanced by the following changes:

- For tasks declared within a generic package, allow task priority to be specified as a generic parameter. This would have been useful for the 1553B bus interfaces of the 11th Missile Application. One of the 1750A processors has two bus interfaces; the code for the interfaces was identical except for task priorities and command port addresses.
- Allow task priorities to be specified dynamically. This would be particularly useful for multi-window user-interface software, in which an instantiation of a generic task is created for each window. The task that is currently interacting with the user would then be able to elevate its priority (or have it elevated).

4. ADDRESS CLAUSES

Address clauses allow objects to be tied to specific addresses (for example, I/O ports). Currently, the Ada language standard requires that an address clause be "immediately within the same declarative part, package specification, or task specification" as the object it references. The address is required to be a *simple* expression.

Reusability would be enhanced by the following changes:

- Allow addresses to be specified as a generic parameter. As noted in the previous section, the 1553B bus interfaces in the 11th Missile Application were identical except for task priorities and command port addresses.
- Allow the address of a specification item to be specified in the body. This would allow a package (or task) specification to have multiple bodies, with each body mapping items declared in the

specification to addresses as required. Since the specification would not be affected, the code using the package (or task) would not have to be recompiled.

5. IMPLEMENTATION OF REDUCED-PRECISION FLOATING POINT TYPES

The use of strong data typing in real-time embedded applications supports reusability but at a cost to the developer and user of reusable software in these applications. As detailed in Section VI, these applications require large numbers of mathematical operators for the different types of data. This subsection discusses the inadequacy of the Ada standard in addressing the needs of applications with large numbers of operators.

The application developer has two choices in maintaining strong data typing in a real-time embedded system:

- Create a few base types and declare subtypes. The strong data typing results from establishing and enforcing rules in the use of the subtypes. This implicit form of strong data typing must be imposed because there will be no matching between parameters in assignments or subprograms other than that of range. If two objects have the same base type they can be treated as if they are of the same type, unless an object assignment is out of range.

The lack of strong type checking simplifies the end-user's job. The user does not have to create large numbers of operators to deal with the subtypes unless operations are between different base types; however, the user must understand that any operations on subtypes will be those of the base type. For example, a subtype which restricts the range and precision of a double precision base to single precision will have the same operators as the base type so the user will get double precision operators on single precision subtypes.

- Create a few parent types and declare derived types. Derived types prevent any operations between types other than those explicitly declared or existing as derived operations. The strong data typing rules are explicitly enforced. The user of derived types must create his own operators for operations between these types, whether or not they are derived from the same parent.

CAMP supports the user of derived types by supplying many of the operations between data types which are likely to occur in applications using the CAMP parts. This reduces the need for user-created operators, assuming the CAMP operators meet his requirements. The CAMP parts cannot, however, address the low-level operator needs of all end-users. In particular, real-time embedded applications frequently mix single and double precision data types for similar objects. For example, there may be multiple measurements of acceleration, e.g., gravitational acceleration, missile vertical acceleration, missile horizontal acceleration. These objects may differ in precision; thus, to strictly maintain strong data typing, they should be of different types, though all are derived from the same parent.

In general, CAMP parts do not account for this strong data typing where precision is the discriminat-

ing feature. It is a problem both for subtypes, where the user obtains the same operator regardless of the precision of the subtype, and for derived types, where the user is required to create operators which the CAMP parts cannot provide. To explicitly support derived types of all precision would require an explosion in the number of generic formal types and generic formal subprograms, plus the number of predefined types and operators to be used as actual parameters in instantiations. The user, in this case, must either use subtypes and enforce strong data typing implicitly through strict management of the development process, or must create his own operators.

Compilers may not allow a choice of operators for derived types. The compiler can *legally* generate code such that all numerical operations result in the generation of code of the highest precision for the target machine. The Ada standard places conformance requirements on the final result of a computation, not on intermediate results.

Ada does not provide any support for the use of subtypes to account for different precisions. Mathematical operations on a subtype are exactly those of the base type (Reference 8, pp 3-23, Section 3.5.3, paragraph 16). In order to modify the precision of the operation generated for the target, the user must perform explicit type conversions. Of course, this will work only if the compiler generates internal mathematical operations based on the precisions of the types.

An obvious solution to this problem is to eliminate the restriction from the Ada standard that operations on subtypes be those of the base type. It would be possible for a compiler to recognize subtype attributes and generate code to match. For example, Figure 37 shows a double precision base type and double and single precision subtypes. Ada will permit mixing of these types in operations but all operations will be double precision. The recommended language change would generate code such that the operations follow the precision of the result type.

```

package Basic_Types is

  Double : constant := 9;
  Single : constant := 6;

  type Double_Precisions is digits Double;

  subtype Single_Precisions is Double_Precisions digits Single;

  -- -- Double precision operations

  function "*" (Left: Double_Precisions;
               Right: Single_Precisions) return Double_Precisions;

  -- -- Single precision operations

  function "*" (Left: Double_Precisions;
               Right: Single_Precisions) return Single_Precisions;

end Basic_Types;

```

Figure 37. Subtypes Should Support Reduced-Precision Operations

6. PROCEDURAL DATA TYPES

At the present time, Ada has no facility for defining procedural data types. As a result, subprograms (procedures and functions) cannot be passed as parameters. There are, however, two contexts in which this capability is not only desirable, but practically indispensable. The first context is state machine applications where the states and transitions are either dynamic or unknown at compile-time; the second context is that of artificial intelligence (AI) applications.

In the state machine context, the user often wishes to be able to dynamically control states of an application, adding or subtracting states as needed. The inability to define procedural data types presents a considerable handicap since this requires all procedures to be known at compile-time; any time a new state and transition needs to be added or deleted, the whole state machine must be recompiled. The ability to pass subprograms as parameters would allow an application to dynamically specify transitions and actions associated with new states. User interface systems often fit this category of applications. In a finite state machine, where the number of states remains fixed, often the actions associated with states or transitions need to change dynamically. The ability to pass subprograms would make possible this type of dynamic allocation for these applications as well as in more general state machines.

In the area of AI systems, the ability to pass subprograms as parameters is also highly desirable. Because artificial intelligence applications rely heavily "on the ability to use procedures as storable, denotable objects" (Reference 9), the lack of this ability in Ada considerably diminishes the capability to express AI paradigms.

7. DYNAMIC BINDING OF BODIES TO SPECS

Currently, only a one-to-one relationship between package bodies and specifications is permitted by the Ada language. In most instances, this is sufficient, but there are cases where a many-to-one relationship would be useful. This would allow multiple package bodies for a single package specification to exist simultaneously in the same Ada library. Which body should be used by the compiler for code generation would be specified when the user imported the package via a *with* statement.

One instance where this would be useful is when working with the CAMP Standard_Trig package; a partial specification for the Standard_Trig TLCSC is shown in Figure 38. This package defines a set of standard trigonometric operations for a system. In order to implement the supplied functions, the package body of Standard_Trig instantiates portions of the Polynomials package.

Figure 39 contains a partial package specification for the Polynomials TLCSC. This package contains a large number of polynomial solutions to various transcendental functions. It also provides access to the transcendental functions provided by the VAX Ada environment.

During the CAMP parts development effort, the package body of Standard_Trig instantiated portions of the Polynomials.System_Functions LLCSC in order to obtain access to the VAX-supplied transcendental functions (see Figure 40). While this package body would be useful for anyone doing development using VAX Ada, it would not be appropriate for an application designed to run in an embedded environment. A modification that would allow this part to be used in an embedded environment would involve

**THIS REPORT HAS BEEN DELIMITED
AND CLEARED FOR PUBLIC RELEASE
UNDER DOD DIRECTIVE 5200.20 AND
NO RESTRICTIONS ARE IMPOSED UPON
ITS USE AND DISCLOSURE.**

DISTRIBUTION STATEMENT A

**APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.**

```

generic
  type Angle      is digits <>;
  type Trig_Ratio is digits <>;
  Pi_Value       : in Angle;
package Standard_Trig is
  type Radians      is new Angle;
  type Sin_Cos_Ratio is new Trig_Ratio range -1.0 .. 1.0;
  type Tan_Ratio     is new Trig_Ratio;
  function Sin (Input : Radians) return Sin_Cos_Ratio;
  function Cos (Input : Radians) return Sin_Cos_Ratio;
  function Tan (Input : Radians) return Tan_Ratio;
end Standard_Trig;

```

Figure 38. Partial Standard_Trig Package Specification

```

package Polynomials is

  package Hastings is

    generic
      type Radians      is digits <>;
      type Sin_Cos_Ratio is digits <>;
    package Hastings_Radian_Operations is
      function Sin_R_4term (Input : Radians) return Sin_Cos_Ratio;
      function Sin_R_5term (Input : Radians) return Sin_Cos_Ratio;
    end Hastings_Radian_Operations;

  end Hastings;

  package System_Functions is

    generic
      type Radians      is digits <>;
      type Sin_Cos_Ratio is digits <>;
    package Radian_Operations is
      function Sin (Input : Radians) return Sin_Cos_Ratio;
    end Radian_Operations;

  end System_Functions;

end Polynomials;

```

Figure 39. Partial Polynomials Package Specification

selecting polynomial solutions from the Polynomials TLCSC and instantiating them accordingly. If the application required the selection of algorithms for multiple precisions, the required modifications would be more extensive since the Standard_Trig package has been designed to provide overloaded operations for different units, but not for different precisions.

```

with Polynomials;
package body Standard_Trig is

    package Radian_Opns is new Polynomials.System_Functions.Radian_Operations
        (Radians => Radians,
         Sin_Cos_Ratio => Sin_Cos_Ratio);

    function Sin_R (Input : Radians)
        return Sin_Cos_Ratio renames Radian_Opns.Sin;

    function Sin (Input : Radians) return Sin_Cos_Ratio is
    begin
        return Sin_R (Input => Input);
    end Sin;

end Standard_Trig;

```

Figure 40. System Functions Version of Standard_Trig Package Body

Under the current definition of the Ada language, the problem of modifying the Standard_Trig package to provide trigonometric functions for multiple precisions could be solved in one of the following ways:

1. Duplicate packages could be provided for each precision. This is the approach that was taken on the 11th Missile Application. It involved duplicating package specification code, giving each package its own unique name, and then implementing the bodies for the different precisions (see Figures 41 and 42). This method has the disadvantage of requiring the creation of packages which are identical except for their names.

```
with Polynomials;
package body Standard_Trig is
  package Radians_Opns is new Polynomials.Hastings.Hastings_Radian_Operations
    (Radians => Radians,
     Sin_Cos_Ratio => Sin_Cos_Ratio);
  function Sin_R (Input : Radians)
    return Sin_Cos_Ratio renames Radians_Opns.Sin_R_4Term;
  function Sin (Input : Radians) return Sin_Cos_Ratio is
  begin
    ...
    return Sin_R (Input => Input);
  end Sin;
end Standard_Trig;
```

Figure 41. Single Precision Version of Standard_Trig Package Body

```
with Polynomials;
package body Standard_Trig is
  package Radians_Opns is new Polynomials.Hastings.Hastings_Radian_Operations
    (Radians => Radians,
     Sin_Cos_Ratio => Sin_Cos_Ratio);
  function Sin_R (Input : Radians)
    return Sin_Cos_Ratio renames Radians_Opns.Sin_R_5Term;
  function Sin (Input : Radians) return Sin_Cos_Ratio is
  begin
    ...
    return Sin_R (Input => Input);
  end Sin;
end Standard_Trig;
```

Figure 42. Extended Precision Version of Standard_Trig Package Body

2. The package specification for Standard_Trig could be modified to allow for multiple precisions as shown in Figure 43. This method is less desirable than Solution 1 because it requires major modifications to the package specification, as well as creation of a new body.
3. The baseline CAMP Standard_Trig package could be modified in a manner similar to that discussed in Solution 2 and shown in Figure 43. This approach has the following disadvantages:
 - It requires the user to instantiate the package and possibly receive code for multiple precisions even if only one precision was required.

```

generic
  type Angle_Single_Precision      is digits <>;
  type Angle_Extended_Precision    is digits <>;
  type Trig_Ratio_Single_Precision is digits <>;
  type Trig_Ratio_Extended_Precision is digits <>;
  Pi_Value_SP                      : in Angle_Single_Precision;
  Pi_Value_EP                      : in Angle_Extended_Precision;
package Standard_Trig is

  type Radians_SP      is new Angle_Single_Precision;
  type Radians_EP      is new Angle_Extended_Precision;

  type Sin_Cos_Ratio_SP is new Trig_Ratio_Single_Precision range -1.0 .. 1.0;
  type Sin_Cos_Ratio_EP is new Trig_Ratio_Extended_Precision range -1.0 .. 1.0;

  type Tan_Ratio_SP      is new Trig_Ratio_Single_Precision;
  type Tan_Ratio_EP      is new Trig_Ratio_Extended_Precision;

  function Sin (Input : Radians_SP) return Sin_Cos_Ratio_SP;
  function Sin (Input : Radians_EP) return Sin_Cos_Ratio_EP;

  function Cos (Input : Radians_SP) return Sin_Cos_Ratio_SP;
  function Cos (Input : Radians_EP) return Sin_Cos_Ratio_EP;

  function Tan (Input : Radians_SP) return Tan_Ratio_SP;
  function Tan (Input : Radians_EP) return Tan_Ratio_EP;

end Standard_Trig;

```

Figure 43. Multiple Precision Version of Standard_Trig Package Specification

- It does not solve the problem of what to do if more than two precisions are required.

The preferred solution would permit a single package specification to have multiple bodies. The Standard_Trig package specification code would then not require modification; the user would create multiple bodies, all of which would exist in the program library at the same time, and then specify which body was to be used at the time the Standard_Trig package was imported. This approach has the advantage of increasing the reusability of the Standard_Trig package specification since it would require no modifications, regardless of the number of bodies required. It also decreases the effort to use the part.

8. SEPARATION OF REPRESENTATION CLAUSES

Flexibility and ease of use are key attributes of good reusable software. Ideally, the design of a reusable part should be sufficiently flexible to permit tailoring without modification of the source code itself. If source code modifications are required, modification of the body is preferable to modification of the specification since this increases the reusability of the specification, avoids modification of the external interface provided by the specification, and potentially eliminates the need to recompile other portions of the system which are dependent upon the modified part. One aspect of tailoring which cannot be accounted for through good design is the need for representation clauses.

"Representation clauses specify how the types of the language are to be mapped onto the underlying machine. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware)" (Reference 8, pp 13-1). One

example of the use of representation clauses in the 11th Missile Application is in the definition of messages sent across the data bus and in size specifications for objects of certain types. An example of this is shown in Figure 44 where the contents and storage representation of a BIM_Error_Message are defined. A further explanation of the contents and storage representation of this message can be found in Table 6. The current definition of the Ada language states that "a representation clause and the declaration of the entity to which the clause applies must both occur immediately within the same declarative part, package specification, or task specification" (Reference 8, pp 13-1). While the need for representation clauses can be anticipated, their form cannot be since they are application-specific. As a result, whenever a user wishes to apply a representation clause to an entity defined in the package specification of a reusable part, the source code for the specification must be modified.

```

with BIM_Interface;
with Bus_Terminals;
with Representation_Parameters;
package Bus_Messages is

  package BI renames BIM_Interface;
  package BT renames Bus_Terminals;
  package RP renames Representation_Parameters;

  type Dummy_Array is array (BI.Word_Counts range <>) of BI.Data_Words;

  type Short_Boolean      is new BOOLEAN;
  for Short_Boolean'SIZE use 1;

  type Short_Feast       is new INTEGER range -(2**15)..(2**15)-1;
  for Short_Feast'SIZE use 16;

  BIM_Error_Word_Count : constant BI.Word_Counts := 1;
  BIM_Error_Dummy_Size : constant BI.Word_Counts
                        := BI.Message_Size_Words -
                           BIM_Error_Word_Count;

  type BIM_Error_Messages is
    record
      Word_Count      : BI.Word_Counts;
      Source           : BT.Terminals;
      Destination      : BT.Terminals;
      Message_Number   : BI.Message_Numbers;
      Status           : BI.Status_Words;
      Dummy            : Dummy_Array(1..BIM_Error_Dummy_Size);
    end record;

  for BIM_Error_Messages use
    record
      Word_Count      at 0 *RP.Storage_Units_per_Word range 0..15;
      Source           at 1 *RP.Storage_Units_per_Word range 0..3;
      Destination      at 1 *RP.Storage_Units_per_Word range 4..7;
      Message_Number   at 1 *RP.Storage_Units_per_Word range 8..15;
      Status           at 2 *RP.Storage_Units_per_Word range 0..15;
      Dummy            at 3 *RP.Storage_Units_per_Word
                        range 0..BIM_Error_Dummy_Size *
                           RP.Message_Word_Size-1;
    end record;

end Bus_Messages;

```

Figure 44. 11th Missile Application Use of Representation Clauses

TABLE 6. BIM_ERROR_MESSAGES CONTENTS AND STORAGE REPRESENTATION

Message Component	Description	Storage Representation
Word_Count	Number of words in the message	Located in bits 0-15 of the word offset 0 words from the beginning of the data structure
Source	Where the message came from	Located in bits 0-3 of the word offset 1 word from the beginning of the data structure
Destination	Where the message is being sent	Located in bits 4-7 of the word offset 1 word from the beginning of the data structure
Message_Number	Used to distinguish between messages of the same type	Located in bits 8-15 of the word offset 1 word from the beginning of the data structure
Status	The error itself	Located in bits 0-15 of the word offset 2 words from the beginning of the data structure
Dummy	A 'filler' array used to keep the overall size of all messages the same	Starts in bit 0 of the word offset 3 words from the beginning of the data structure and continues for the number of words required to completely fill the message structure

Allowing representation clauses to be specified in a package body for entities declared in the package specification would increase the reusability of the specification. In cases where the reusable software consists of only the package specification with the body being user-defined, a representation clause could be defined in the Ada body without any modifications to the reusable specification code. In cases where a body already existed, permitting the modification to be made to the body instead of the specification could potentially eliminate the need to recompile other portions of the system. The placement of representation clauses in package bodies would be consistent with the specification (Ada data structure) versus body (implementation) split which exists throughout the Ada language.

SECTION VI

PARTS DESIGN METHODOLOGY

1. DESIGN REQUIREMENTS

The development of reusable software presents the designer with a conflicting set of design requirements. In addition to being reusable on a number of different real-time embedded applications, the design of reusable parts must address the following issues:

- Well-defined interfaces
- Efficient implementations
- Strong data typing to minimize inappropriate use
- Availability of mathematical operations on different data types
- Simplicity of use

These conflicting requirements have led some to the conclusion that "since the generality needed for flexibility and portability will increase software overhead and, consequently, decrease the software's efficiency . . . it is very difficult to construct reusable missile software that is still viable." (Reference 10, p 105). This is not the conclusion from the CAMP project.

The CAMP project developed a design methodology to address these conflicting issues and produce reusable parts for missile applications. A set of reusable part goals — flexibility, efficiency, ease of use, and protection against misuse — form the basis of this method. *Flexibility* is the extent to which parts can be modified or tailored to the specific needs of an individual application. Although parts may be reusable, if they are not flexible and easily tailored, then the cost of using a part may be prohibitively large and it may be less expensive to develop a new part than to try to tailor or modify an existing part. The issue of *efficiency* is one which has long plagued reusability: the contention is that parts which are reusable can never attain the required efficiency for use in real-time embedded applications. The parts must also be *easy to use* because difficulty of use increases cost of reuse and may mean that the part will never be reused. *Protection against misuse* refers to providing the user with protection from choosing the wrong part for a given requirement or calling the part with improper parameters. The use of the Ada *generic* feature and strong data typing prevents misuse to some extent. However, there are other features which may be included in the design of parts to guard against misuse.

The CAMP design methodology meets the reusability goals and supports parts which are well-tested and may be used off-the-shelf. The methodology utilizes several of Ada's special, though standard, programming features, including derived types and subprograms, generic instantiation and subprogram overloading. The CAMP team believes that this design approach will be equally appropriate for non-missile applications outside the CAMP domain.

2. DESIGN METHODS

Six methods for the design of reusable parts were considered on the CAMP program. Figure 45 illustrates these methods. In discussing the competing methods, a specific CAMP part will serve as an example. This part, `Compute_Earth_Relative_Horizontal_Velocities`, has three inputs:

- Nominal_East_Velocity (VEL_{NE})
- Nominal_North_Velocity (VEL_{NN})
- Wander_Angle (WA)

It processes these inputs through the following equations:

- $VEL_E := VEL_{NE} * \cos(WA) - VEL_{NN} * \sin(WA)$
- $VEL_N := VEL_{NN} * \cos(WA) + VEL_{NE} * \sin(WA)$

producing the following outputs:

- True East Velocity (VEL_E)
- True North Velocity (VEL_N)

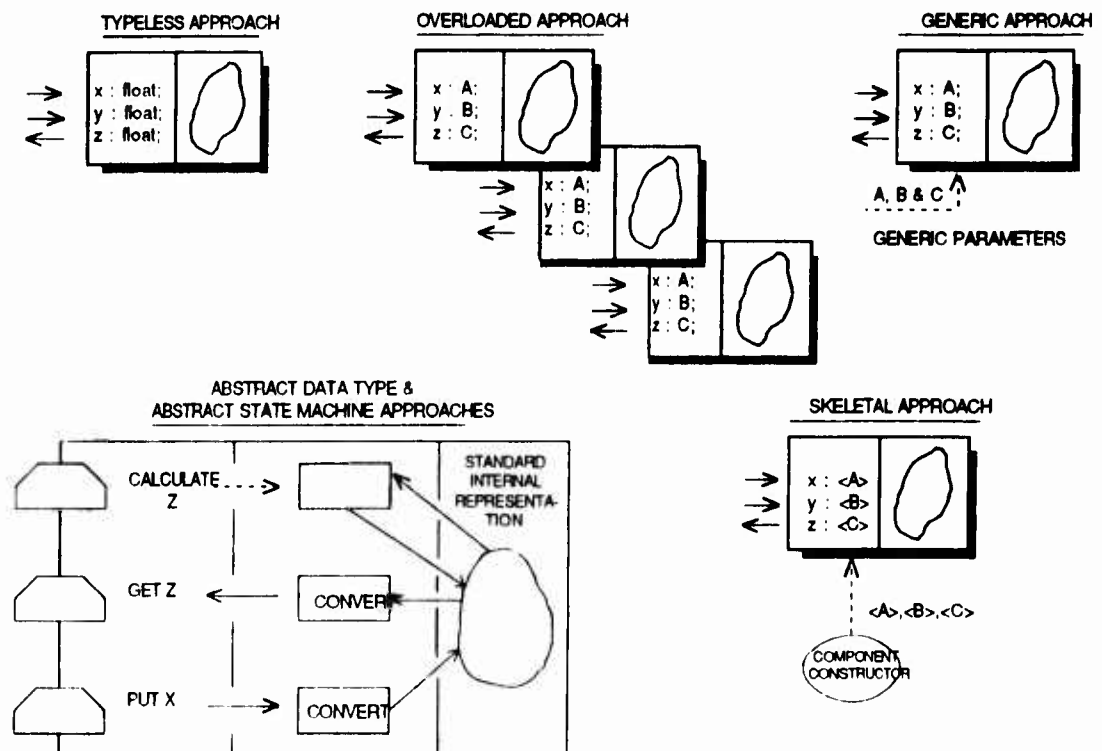


Figure 45. Reusable Parts Methods

The computations performed in this example require trigonometric functions on `Wander_Angle` plus multiplication and subtraction operators. In addition, the multiplication operator must perform its opera-

tion on data of different types, namely, a velocity type and a sine- or cosine-of-an-angle type. These mathematical functions must also be provided for all possible combinations of data types for velocities and angles. For example, if velocity is measured in feet-per-second and angle is in radians, the following mathematical operations are required:

- Sine and cosine operations on radians;
- Multiplier for feet-per-second by the result of the sine and cosine operations; and,
- Subtractor for the result type of the multiplier.

The discussion will explain the methods for parts design and evaluate their effectiveness with regard to the following four evaluation criteria:

1. Efficiency and appropriateness of the interface;
2. Control for preventing misuse;
3. Availability of needed mathematical operators and functions; and,
4. Degree to which the user's job is simplified.

Following the presentation of the six methods, this section will focus on the method chosen on the CAMP program for parts design, and the implications of that method for a generalized parts development environment.

a. Typeless Method

The *typeless* method assumes that all data objects and actual parameters will be of the universal float type. The benefit of this approach is to alleviate the need for special mathematical operators and functions since they are already defined in standard packages. The severe disadvantage is that the compiler and runtime system cannot perform type checking to prevent misuse of the part.

The failure of an SDI-related experiment in 1985 illustrates the problems which can result without strong typing. The experiment required an orbiting receiver to track a ground-based laser. The transmitter was positioned at an elevation of approximately 10000 feet and this elevation was to be entered into the flight computer of the receiver's orbiting platform. The flight computer was programmed to accept ground elevation in *nautical miles* not feet, however, so when 10000 was entered, the platform oriented the receiver to point to a position 10000 miles above the surface of the earth, exactly 180 degrees from the correct location, 10000 *feet* above the surface.

Strong typing of parameters could alleviate this problem. Figure 46 illustrates the data type and object declarations which would apply.

```

type Nautical_Miles is digits 6;

subtype Ground_Elevation is Nautical_Miles range -1.0 .. 6.0;

Transmitter_Elevation : Ground_Elevation;

```

Figure 46. Strong Data Typing Example

This would restrict the input values of Transmitter_Elevation to a reasonable range for units of nautical miles.

The specification shown in Figure 47 illustrates the typeless method. A user application accessing this procedure could pass any object of the type float as actual parameters. The compiler could not perform type checking to prevent possible type mismatching and there could be no runtime checking to assure correct ranges for the actual parameters. This method produces parts which are easy to use, but offers no protection against misuse.

```

procedure Compute_Earth_Relative_Horizontal_Velocities
(Nominal_East_Velocity : in    FLOAT;
 Nominal_North_Velocity : in    FLOAT;
 Wander_Angle          : in    FLOAT;
 East_Velocity          : out   FLOAT;
 North_Velocity         : out   FLOAT);

```

Figure 47. Typeless Method

b. Overloaded Method

To allow a greater choice in data typing, the *overloaded* method provides the user a separate version of each part to allow for the different combination of data types which the part user might require. The code segment shown in Figure 48 illustrates the overloaded method applied to the example when the velocities are of type Feet_Per_Second and Meters_Per_Second and Wander_Angle is in Radians.

```

package Overloaded_Method is

  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocity : in    Feet_Per_Second;
     Nominal_North_Velocity : in    Feet_Per_Second;
     Wander_Angle          : in    Radians;
     East_Velocity          : out   Feet_Per_Second;
     North_Velocity         : out   Feet_Per_Second);

  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocity : in    Meters_Per_Second;
     Nominal_North_Velocity : in    Meters_Per_Second;
     Wander_Angle          : in    Radians;
     East_Velocity          : out   Meters_Per_Second;
     North_Velocity         : out   Meters_Per_Second);

end Overloaded_Method;

```

Figure 48. Overloaded Method

Other overloaded subprograms would allow `Wander_Angle` in degrees and semicircles. This is the method used by Ada packages such as `STANDARD`, `CALENDAR`, and `TEXT_IO` to provide identical operations on different data types.

The overloaded method offers the protection of strong data typing with simplicity of design and use of parts. The designer will decide which combinations of data types to allow for each part and will explicitly declare the parameter interfaces for each overloaded subprogram. He will also define all of the mathematical parts which the subprograms will use: sine and cosine for `Wander_Angle`, and the multiplication and subtraction operators. Strict type checking will assure that actual and formal parameters match and that the values of the actual parameters fall within ranges allowed by the type.

Because Ada supports this overloading of subprogram definitions, the user need not call a version of a part specific to a given combination of data types; the Ada disambiguation feature will resolve the call. In fact, should user requirements change and a different combination of data types result, the call need not be changed, the Ada language will resolve the new reference. This method therefore provides simplicity of use with the protection associated with strong data typing.

The major disadvantage of this method is the large number of parts declared at the architectural level. For the data types stated above (`Feet_Per_Second` and `Meters_Per_Second` for velocities and `Radians`, `Degrees`, and `Semicircles` for angles), the `Compute_Earth_Relative_Horizontal_Velocities` procedure would require six specifications and bodies to accommodate the different combinations of data types. A navigation package encapsulating a complete set of reusable navigation parts could easily grow to over 100 subprograms. Thus, the overloaded method, while simple to use, would be almost impossible to develop and maintain.

c. Generic Method

The *generic* method uses Ada generic units to provide parts which are tailorable to user-defined data types. Figure 49 shows the generic method applied to the `Compute_Earth_Relative_Horizontal_Velocities` procedure using *generic formal types* for `Velocities`, `Angles`, and `Sin_Cos_Ratio` (the type returned by a call to `Sine` or `Cosine`), and *generic formal subprograms* for the required trigonometric functions and the multiplication operator. The subtraction operator operates only on the generic velocity type and is implicit from the generic definition.

The generic formal subprogram parameters are used within the body of the part to perform mathematical operations on objects of the generic data types. For example, the sine and cosine operations on `Wander_Angle` are performed by the functions supplied as actual parameters for the generic `Sin` and `Cos`. The user must define operators to perform these functions on objects of the actual type for `Angles` returning an object of the `Sin_Cos_Ratio` type. This large number of generic parameters could place an enormous burden on the part user, requiring him to create and supply all of the needed actual parameters, both types and subprograms. For the example, the user must supply three data types plus three subprograms as actual parameters.

A method which uses default parameters could alleviate some of this overhead from the part's user. If the total parts design includes typical data types and provides functions for typical combinations

```

generic
  type Sin_Cos_Ratio is digits <>;
  type Velocities is digits <>;
  type Angles is digits <>;
  with function "*" (Left : Velocities;
                     Right : Sin_Cos_Ratio) return Velocities is <>;
  with function Sin (In_Angle : Angles) return Sin_Cos_Ratio is <>;
  with function Cos (In_Angle : Angles) return Sin_Cos_Ratio is <>;
  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocity : in Velocities;
     Nominal_North_Velocity : in Velocities;
     Wander_Angle : in Angles;
     East_Velocity : out Velocities;
     North_Velocity : out Velocities);

```

Figure 49. Generic Method

of these data types, then the user could provide predefined types as actual type parameters and the actual subprogram parameters will default to the predefined functions. Figure 50 illustrates this method. Using the same example, the design could incorporate a separate data types part supplying a Radians type and trigonometric operations on Radians. The multiplication operator could be similarly predefined. This approach yields a *tunneling* of parameters, where explicit use of a type allows tunneling of operators on that type. The advantage of this method is clear: the user obtains the protection associated with strong data typing and the flexibility of using a choice of data types without the need to define his own types or operators.

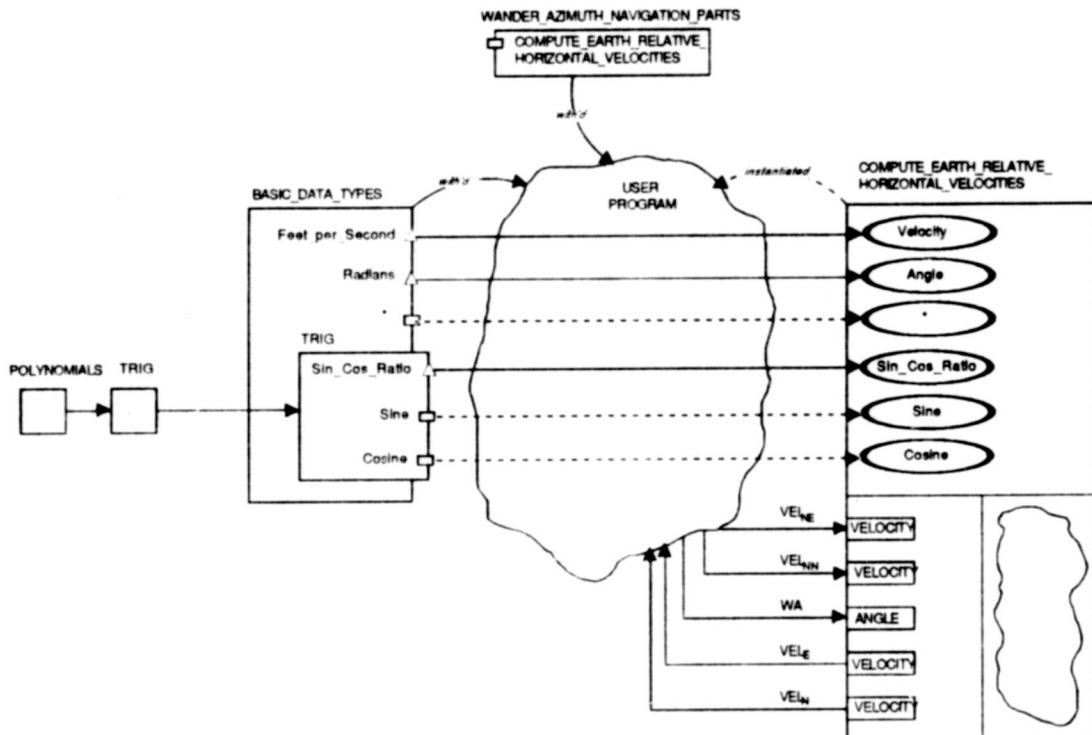


Figure 50. Tunneling of Parameters

d. Abstract State Machine Method

The *abstract state machine* method affords the part user a very high-level interface to reusable parts. In this method, the interface is a package structure defining all of the characteristics of the missile state relevant to a group of navigation operations. In a *state machine*, the interface is strictly through the operations, as the user does not have direct access to or knowledge of the data structure on which the operations work (Reference 11, pp 202). The state machine allows the underlying structure to change without the users' knowledge.

The state machine implementation of a navigation system would provide all of the operations needed to perform the navigation function, both those changing the state and those reporting the state. One such function would be the `Compute_Earth_Relative_Horizontal_Velocities` operation which would both update and report the velocity. Figure 51 contains a code segment to illustrate the abstract state machine method.

```
generic
  type Sin_Cos_Ratio is digits <>;
  type Velocities    is digits <>;
  type Angles        is digits <>;
  type Altitudes     is range <>;
  with function "*" (Left  : Velocities;
                    Right : Sin_Cos_Ratio) return Velocities is <>;
  with function Sin (In_Angle : Angles)    return Sin_Cos_Ratio is <>;
  with function Cos (In_Angle : Angles)    return Sin_Cos_Ratio is <>;
  . . .
package Navigation_State_Machine is

  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocities : in    Velocities;
     Nominal_North_Velocities : in    Velocities;
     Wander_Angle           : in    Angles;
     East_Velocities        : out Velocities;
     North_Velocities       : out Velocities);

  procedure Update_Altitude
    (Vertical_Velocities : in    Velocities;
     Current_Altitude   : out Altitudes);

  -- --operations to provide state information

  function Current_East_Velocities return Velocities;

  function Current_North_Velocities return Velocities;

end Navigation_State_Machine;
```

Figure 51. State Machine Method

The abstract state machine approach utilizes generic units to tailor operations to the user's requirements. Like the generic method, this approach enforces strong data typing and protection against misuse. However, because all operations are encapsulated in a single package, the user is presented with an "all or nothing" solution: specify all of the generic parameters for all operations, whether needed or not, or don't use the package.

An alternative approach would be to encapsulate the data typing and structure within the pack-

age, forcing the part user to convert his typing to conform to that of the abstract state machine. While defining all internal data types and operations makes the part easier to use, the overhead of conversion to the internal data structure would be prohibitive. This conversion would entail not only data typing, but also unit conversion, from meters to feet, radians to degrees, etc. The package could provide interfaces to simplify the unit conversion, but could do little to alleviate the overhead.

The state machine approach does offer an advantage of creating more than one body for a single specification. Because all data is controlled within the body, a part user may use only the specification and write his own body, defining data according to his own choice. Similarly, the parts designer may provide multiple bodies for a single specification, thus alleviating the efficiency issues by creating bodies which are efficient for a particular situation. Like the overloaded method, this increases the cost of creating parts, yet is an effective method when the choice of a data structure cannot, for reasons of efficiency or simplicity, be established in the package specification.

c. Abstract Data Type Method

Like the abstract state machine method, the *abstract data type* method offers the part user a high-level interface to reusable parts. This interface consists of a predefined set of operations on a data structure, and, unlike the abstract state machine, the interface includes the data structure itself. The user, therefore, knows the structure he is dealing with and, depending on the implementation, may even be able to access the structure directly. In the abstract data type method the user is aware of changes to the state of the structure which are effected by the exported operations.

In most implementations, an abstract data type restricts access to objects of the abstract type to operations defined in the package specification. In contrast to the abstract state machine, this type is part of the specification, and the package body must operate on that unique structure. If a part user wishes to use the operations of the abstract data type but use a different data structure, then he must not only rewrite the body which will operate on the data structure, but also rewrite the specification which defines the structure. This method is used extensively in abstract data structures such as vectors and matrices, stacks and queues, but is less appropriate for more complex data structures such as those used by a navigation system or Kalman filter.

A package which implements the navigation system according to the abstract data type method looks quite similar to that of the abstract state machine. (See Figure 52.) The major distinction is the **private** section of the specification which defines the abstract data structure.


```

generic
  type Sin_Cos_Ratio is digits <>;
  type Velocities    is digits <>;
  type Angles        is digits <>;
  type Altitudes     is range <>;
  with function "*" (Left : Velocities;
                     Right : Sin_Cos_Ratio) return Velocities is <>;
  with function Sin (In_Angle : Angles)      return Sin_Cos_Ratio is <>;
  with function Cos (In_Angle : Angles)      return Sin_Cos_Ratio is <>;
package Navigation_State_Machine is

  type Navigation_Model is private;

  procedure Update_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocity : in Velocities;
     Nominal_North_Velocity : in Velocities);

  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Updating : in out Navigation_Model);

  procedure Update_Vertical_Velocity
    (Vertical_Velocity : in Velocities);

  procedure Compute_Altitude
    (Updating : in out Navigation_Model);

  -- --operations to information from data structure

  function Current_East_Velocity
    (Based_On : Navigation_Model) return Velocities;

  function Current_North_Velocity
    (Based_On : Navigation_Model) return Velocities;

private -- Definition of Navigation Abstract Data Structure

  type Navigation_Model is
    record
      Missile_Velocity : Velocities;
      Missile_Altitude : Altitudes;
      . . .
    end record

end Navigation_State_Machine;

```

Figure 52. Abstract Data Type Method

This method is similar to the abstract state machine approach in that it utilizes generic units to tailor operations to the user's requirements. It uses these generic units to enforce strong data typing and protect against misuse. However, the user is again presented with an "all or nothing" solution. Again, the abstract data type offers an alternative approach which would define data types in the package specification, eliminating all of the generic units. While defining all internal data types and operations will ease the use of the part, the overhead of conversion to the internal data structure would be prohibitive.

f. Skeletal Code Method

The *skeletal code* method provides the part user with code templates, which may be manipulated in an editor or through some other tool. This approach gives the part user the flexibility of generic units, without the complexity of the generic instantiation. A sample template, as shown in Figure 53 for the Compute_Earth_Relative_Horizontal_Velocities, would look similar to the code for the typeless method.

```
procedure Compute_Earth_Relative_Horizontal_Velocities
  (Nominal_East_Velocity : in ____;
   Nominal_North_Velocity : in ____;
   Wander_Angle         : in ____;
   East_Velocity         : out ____;
   North_Velocity        : out ____);
```

Figure 53. Skeletal Code Template Method

This approach would add complexity by requiring the part user to complete much of the environment. Outside the part, he must edit the skeletal code into his existing design, inserting data types and overloaded operators as required. While the generic method provides a generic specification, and forces conformity through the Ada generic matching rules, the skeletal method can only provide user documentation to support creation of the environment. In addition, if two or more designers are using similar parts, they may choose different values for completing the templates, duplicating parts of the environment. There would also be a tendency to avoid strong data typing to alleviate the overhead attached to creation of overloaded operators and functions.

An expert system, interfacing to the code templates, could support use of the skeletal code method. The expert system could prompt the user for information it needs to fill in the blanks, but rules, stored in the expert system knowledge base, would allow the system to complete the environment, filling in additional types, operators, and any additional subprograms. The expert system approach offers the long-term solution to the difficulties of the skeletal method by building the environment as a by-product of the user dialogue.

3. USE OF THE GENERIC METHOD

The CAMP program conducted a thorough analysis of each method for design of reusable parts. Figure 54 summarizes the results of this analysis and compares advantages and disadvantages of methods.

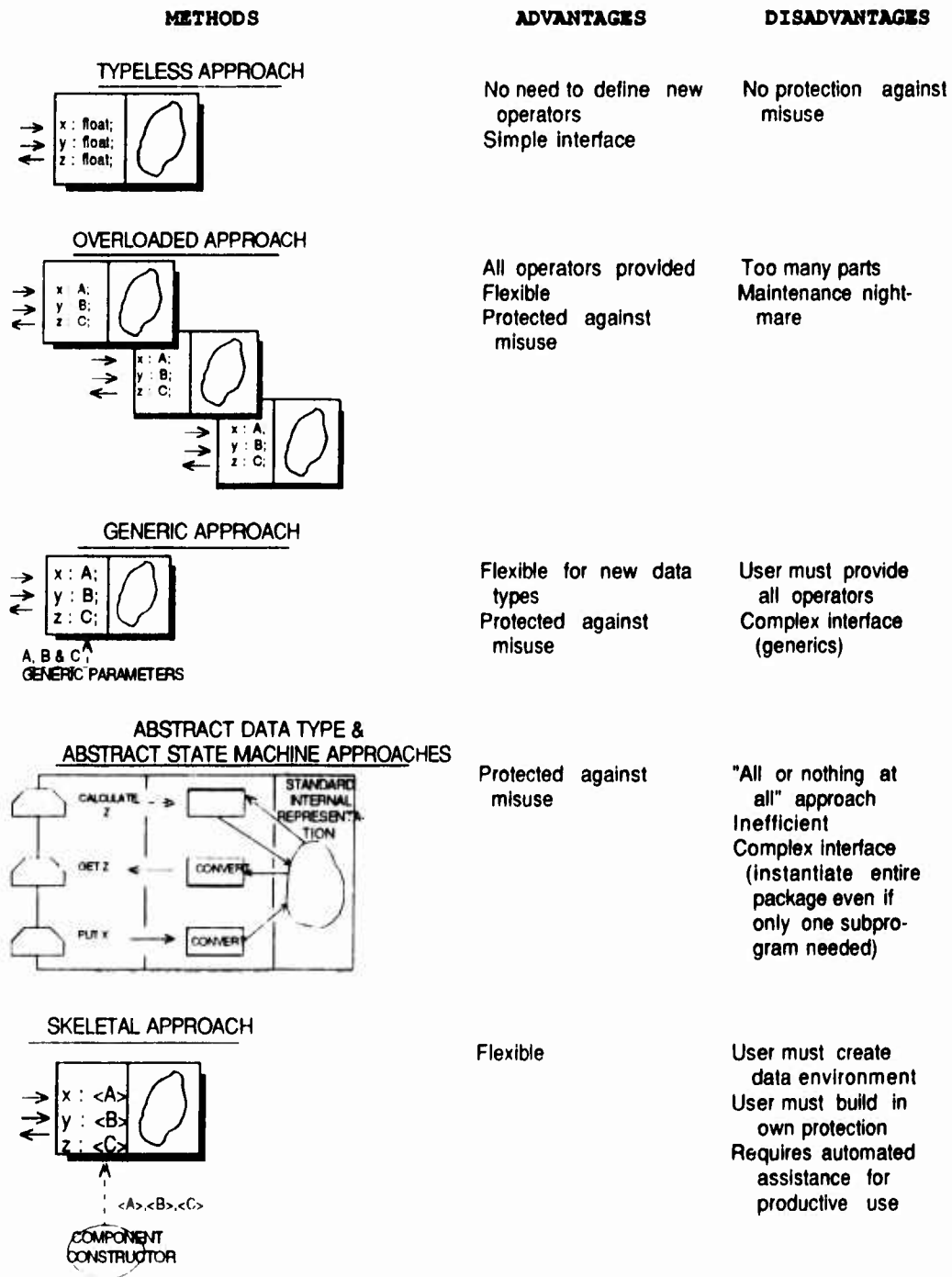


Figure 54. Comparison of the Six Reusable Parts Methods

The analysis focused on the generic method as providing the greatest potential for the design of reusable parts. Prior Data Sciences, a Canadian firm specializing in the development of reusable, real-time software, has summarized the difficulties in developing reusable software based on generic units and of employing parts created using generic units.

- *"Library generic units are very difficult to write . . . the effort required to properly generalize them is usually significant."*
- *"Generic units are also difficult to use, especially when they have many interrelated parameters. The parameter matching rules can be very subtle."* (Reference 12, p 70)

Although generic units add complexity to the interfacing mechanism, the flexibility and protection against misuse which they afford weigh heavily in their favor. Generic units also provide flexibility for tailoring to the requirements of a specific application.

The CAMP parts development team conducted an analysis to determine the best methods for support of complex operators inside the body of parts and for simplification of the use of parts developed using the generic method. The CAMP project has been unique in its investigation of these areas. Most reusability studies have focused primarily on abstract data types, which require only simple generic operators, e.g., integer incrementation, data structure iterators, etc. While some reusability efforts have addressed the needs of the scientific and engineering communities for mathematical software, the resulting parts support neither strong data typing nor user selection of mathematical operators called internal to the part. The following two subsections address two key issues of the CAMP project:

- The approach developed on CAMP for the design of reusable parts using the generic method; and,
- The use of those parts in constructing an application from reusable software.

a. Using the Generic Method to Design Parts

Effective use of generic units for the creation of reusable parts requires reconciliation between the complexity of the generic specification and the desired ease of use of the part. The presentation of the generic method discussed the conflict. In fact, the conflict entails the same trade-offs as those required to create reusable software: generality vs. efficiency and ease of use.

The CAMP parts fully exploit the Ada generic facility. Low-level parts are designed as generic packages or subprograms. Higher-level parts are built from multiple levels of these generic units. The user supplies actual parameters to instantiate the generic parts and tailor them to his application. The CAMP part architecture, with multiple layers of generic units provides the part user with a broad choice in his selection of parts for an application: he may use low-level parts to implement low-level features of the individual objects of his design or choose high-level parts to themselves serve as objects in his design.

A generic part uses its *generic formal parameters* for tailoring the part to a specific application. The `Compute_Earth_Relative_Horizontal_Velocities` part may be tailored for velocity type (feet per second, meters per second, miles per hour, knots) and for angle type (radians, degrees, semicircles). In addition, the tailoring can extend to the return type of a sine or cosine operator.

In order to complete the tailoring, the part must also allow tailoring for operators essential for the enforcement of strong data typing. Generally, operators are merely overloadings of predefined operations ("+", "-", "*", "/"). For more complex operations, the user must create his own subprograms, such as sine and cosine, filters, matrix operations, etc. For these user-created operations, there are no language-defined constructs and the generic specification cannot fully describe the required operation. Only part documentation and the user's familiarity with the part's internal design can support creation of actual parameters to match the formal generic. Those features of a part which are truly common between applications, and are captured in the body of the part, include:

- the use of generic data types
- the sequence of operations
- data types and operations not parameterized through the generic

Figure 55 shows the use of generic plus non-generic features of a part body. The formal data types and Sin and Cos operations are generic and, hence, tailorable. The multiplication operator is also generic. The subtraction and addition operations are not generic. Of course, the sequence of operations to calculate the output velocities is also non-generic.

```

procedure Compute_Earth_Relative_Horizontal_Velocities
  (Nominal_East_Velocity : in      Velocities;
   Nominal_North_Velocity : in    Velocities;
   Wander_Angle          : in    Angles;
   East_Velocity         :         out Velocities;
   North_Velocity        :         out Velocities) is

  Sin_W_A : Sin_Cos_Ratio;
  Cos_W_A : Sin_Cos_Ratio;

begin

  Sin_W_A := Sin (Current_Wander_Angle);
  Cos_W_A := Cos (Current_Wander_Angle);

  East_Velocity := Nominal_East_Velocity * Cos_W_A -
                    Nominal_North_Velocity * Sin_W_A;

  North_Velocity := Nominal_North_Velocity * Cos_W_A +
                    Nominal_East_Velocity * Sin_W_A;

end Compute_Earth_Relative_Horizontal_Velocities;

```

Figure 55. Commonality Captured in the Generic Part Body

b. Using Parts to Construct an Application

The difficulty of the generic method stems from the large number of data types required by a part and the resulting large number of operators on objects of those types. In the example introduced above, the three data types lead to only three required operators. In addition, part use is further simplified by the defaulting mechanism of Ada generic units. Because the three operators exist for a limited range of data types, the CAMP parts structure can provide default versions for each operator. Now, when the user supplies actual types for his instantiation, the operators can default through the tunneling mechanism depicted in Figure 50. The user may, however, wish to override the system's tunneling of parameters by supplying his own operators. CAMP parts also support overriding defaults by providing a selection of such common operators as trigonometric functions. Figure 56 depicts the mechanism of overriding defaults. Here, the user chooses his own cosine function from the CAMP Polynomials package to override the default from Standard_Trig. The user could also write his own cosine function to override the default. The Ada mechanism to accomplish the default overriding is explained in Section II.

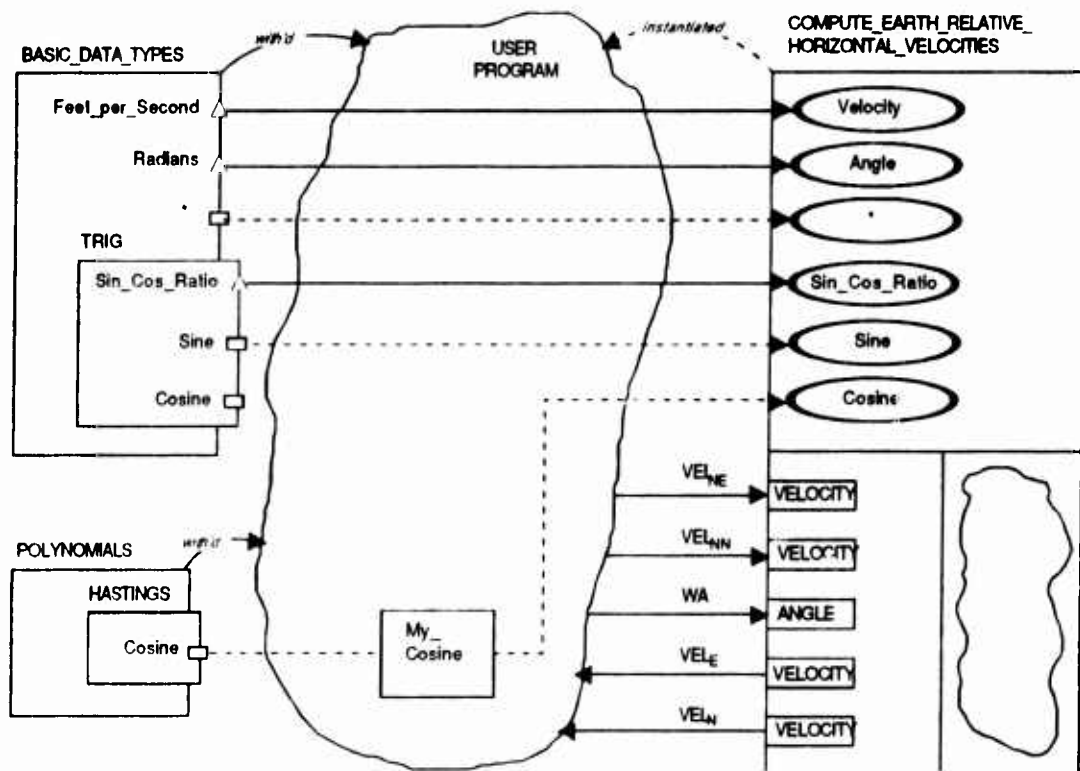


Figure 56. Mechanism for Overriding Defaults

Where the mix of data types and operators grows beyond a manageable level, the need to provide additional assistance to the part user also grows. The generic formal part of the CAMP Lateral/Directional Autopilot (Figure 57), for example, includes:

- twelve formal data types
- ten formal data objects

```

generic
-- --types for Aileron Loop
    type Roll_Commands      is digits <>;
    type Roll_Attitudes     is digits <>;
    type Roll_Command_Gains is digits <>;

-- --types for Rudder Loop
    type Rudder_Cmd_Roll_Rate_Gains is digits <>;
    type Missile_Accelerations      is digits <>;
    type Acceleration_Gains         is digits <>;
    type Gravitational_Accelerations is digits <>;
    type Velocities                 is digits <>;
    type Trig_Value                 is digits <>;

-- --types for both loops
    type Feedback_Rate_Gains is digits <>;
    type Fin_Deflections     is digits <>;
    type Feedback_Rates      is digits <>;

-- --Initial values for aileron control loop
    Initial_Aileron_Integrator_Gain      :
        In Roll_Command_Gains;
    Initial_Aileron_Integrator_Limit     :
        In Fin_Deflections;
    Initial_Roll_Command_Proportional_Gain :
        In Roll_Command_Gains;
    Initial_Roll_Rate_Gain_For_Aileron    :
        In Feedback_Rate_Gains;
    Initial_Yaw_Rate_Gain_For_Aileron     :
        In Feedback_Rate_Gains;

-- --Initial values for rudder control loop
    Initial_Rudder_Integrator_Gain      :
        In Acceleration_Gains;
    Initial_Rudder_Integrator_Limit     :
        In Fin_Deflections;
    Initial_Yaw_Rate_Gain_For_Rudder    :
        In Feedback_Rate_Gains;
    Initial_Roll_Rate_Gain_For_Rudder    :
        In Rudder_Cmd_Roll_Rate_Gains;
    Initial_Acceleration_Proportional_Gain :
        In Acceleration_Gains;

-- --Aileron control loop limiters and filter
    with function Roll_Error_Limit
        (Roll_Command : Roll_Commands)
        return Roll_Commands is <>;

    with function Aileron_Command_Limit
        (Fin_Deflection : Fin_Deflections)
        return Fin_Deflections is <>;

    with function Roll_Command_Filter
        (Roll_Command : Roll_Commands)
        return Roll_Commands is <>;

-- --Rudder control loop limiters, filters, and trig function
    with function Rudder_Command_Limit
        (Fin_Deflection : Fin_Deflections)
        return Fin_Deflections is <>;

    with function Yaw_Rate_Filter
        (Yaw_Rate : Feedback_Rates)
        return Feedback_Rates is <>;

    with function Acceleration_Filter
        (Lateral_Acceleration : Missile_Accelerations)
        return Missile_Accelerations is <>;

    with function Sin (Angle : Roll_Attitudes)
        return Trig_Value is <>;

-- --Aileron control loop gain and updater functions
    with function "-" (Left : Roll_Commands;
        Right : Roll_Attitudes)
        return Roll_Commands is <>;

    with function "*" (Left : Roll_Commands;
        Right : Roll_Command_Gains)
        return Fin_Deflections is <>;

    with function "*" (Left : Feedback_Rates;
        Right : Feedback_Rate_Gains)
        return Fin_Deflections is <>;

-- --Rudder control loop gain and updater functions
    with function "*" (Left : Missile_Accelerations;
        Right : Acceleration_Gains)
        return Fin_Deflections is <>;

    with function "*" (Left : Feedback_Rates;
        Right : Rudder_Cmd_Roll_Rate_Gains)
        return Feedback_Rates is <>;

    with function "*" (Left : Gravitational_Accelerations;
        Right : Trig_Value)
        return Gravitational_Accelerations is <>;

    with function "/" (Left : Gravitational_Accelerations;
        Right : Velocities)
        return Feedback_Rates is <>;

package Lateral_Directional_Autopilot is
    type Aileron_Rudder_Commands is record
        Aileron_Command : Fin_Deflections;
        Rudder_Command : Fin_Deflections;
    end record;

    procedure Initialize_Lateral_Directional_Autopilot
        (Initial_Aileron_Command : In Fin_Deflections;
        Initial_Rudder_Command : In Fin_Deflections;
        Gravitational_Acceleration : In Gravitational_Accelerations;
        Roll_Command : In Roll_Commands;
        Roll_Attitude : In Roll_Attitudes;
        Roll_Rate : In Feedback_Rates;
        Yaw_Rate : In Feedback_Rates;
        Missile_Velocity : In Velocities;
        Lateral_Acceleration : In Missile_Accelerations);

    function Compute_Aileron_Rudder_Commands
        (Roll_Command : In Roll_Commands;
        Roll_Attitude : In Roll_Attitudes;
        Roll_Rate : In Feedback_Rates;
        Yaw_Rate : In Feedback_Rates;
        Lateral_Acceleration : In Missile_Accelerations;
        Missile_Velocity : In Velocities;
        Gravitational_Acceleration : In Gravitational_Accelerations)
        return Aileron_Rudder_Commands;

end Lateral_Directional_Autopilot;

```

Figure 57. Autopilot Part Generic Specification

- seven language-independent operations
- seven language-defined operations

The CAMP design structure eases the burden of the part user by supplying packages of standard data types which may serve as actual types for the generic formal types, packages of standard operators to supply actual subprograms for the generic formal subprogram operations, and a mix of operators overloading the language-defined operations. The user's task is now reduced to *selecting* the proper combination of data types and operators from the parts base. He may create his own, if the CAMP parts base is deficient in some area, but an attempt has been made to cover a high degree of variability. Furthermore, the parts base is easily extended to allow for new standard types and operators. Figure 58 shows the range of selections open to the CAMP parts user.

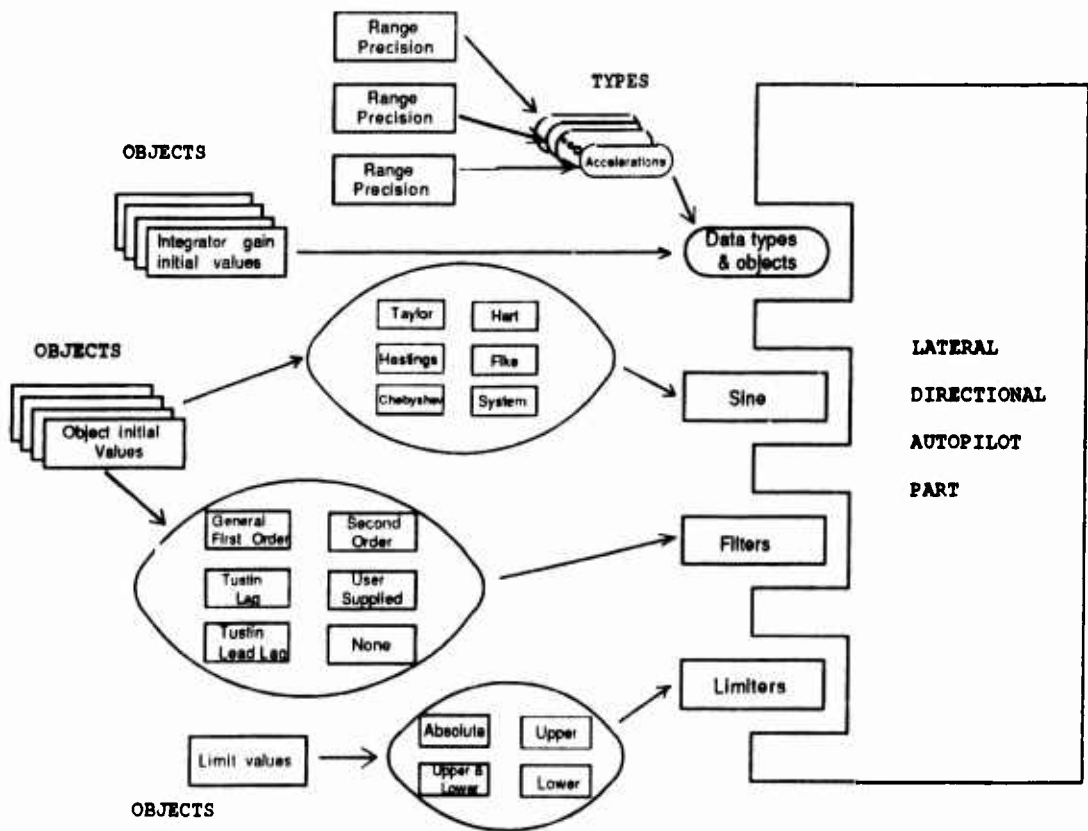


Figure 58. Selections from CAMP Parts for Instantiation

4. SEMI-ABSTRACT DATA TYPE

The combination of high-level parts with lower level support packages providing actual types and operators leads to the creation of a complete environment for use of a part. The CAMP program has established that it is essential to provide support for a complete environment to incorporate reusable software into a design. Others have noted the importance of the environment because "[t]he very concept of reusability must be defined . . . in terms of the dependence of the component on enclosing or higher level environments" (Reference 13, p 550). The CAM method uses the term *part bundle* to describe the environment that consists of a combination of packages required to support a part plus the context clause the user must specify to obtain the environment.

The part bundle allows the user access to a predefined packaging structure. Availability of this structure eases part use by providing the user the environment he needs to use a part. Figure 59 shows the complete bundle required to support the Autopilot package part. In order to use this package, the user must first import the Autopilot part itself. In addition, he needs data types supplied by the Basic_ and Autopilot_Data_Types parts, and signal processing and trigonometric operations supplied by the Signal_Processing and Polynomial parts, respectively. The user is unaware of bundles which exist to support the lower level packages; for example, Signal_Processing bundles General_Purpose_Math, and Basic_Data_Types bundles Standard_Trig, Conversion_Factors, and Universal_Constants.

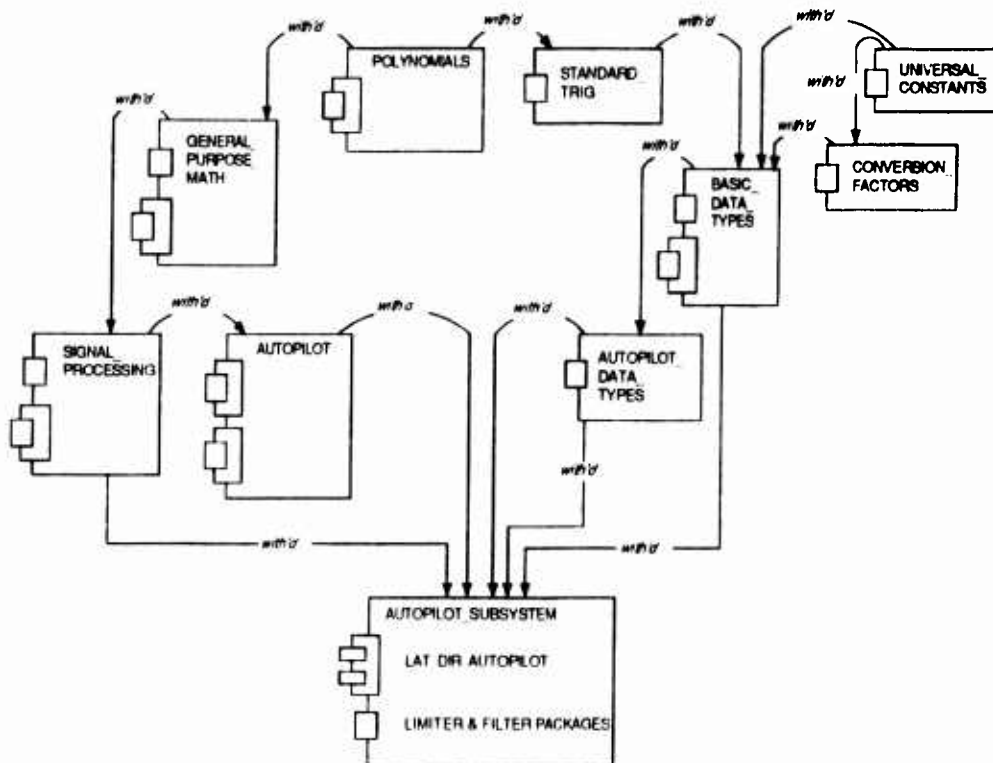


Figure 59. Autopilot Bundle Structure

While the bundle gives the user an environment for use of a part, the user must still extract entities provided by the bundle for tailoring the part to his application. In addition, the user may modify the

bundle, overriding aspects of the bundle by supplying other CAMP parts from CAMP packages or his own parts. This *open architecture* — the ability of the user to supply his own data types and operators — is one of the key design features of CAMP parts use.

The term *semi-abstract data type* is used to formally describe this open architecture of the CAMP generic method. As opposed to the abstract data type which defines an abstract data structure and operations on that structure for its use, the semi-abstract data type is very much under user control.

The use of the Autopilot bundle illustrates the capabilities of the semi-abstract data type. Were the Autopilot part defined as an abstract data type, all data structures and operations would be encapsulated and hidden within the part, with the user tailoring the part through the generic formal parameters. As previously described under the abstract data type method, the user could not gain access to any of the part's facilities, data structures or operations, without going through the part. In contrast, the semi-abstract data type allows the user access to a bundle, which also provides access to all the part's facilities. In addition, the bundle allows access, on an individual basis, to data types from the types packages and to functional parts from the Signal_Processing or Polynomial packages. The user is free to use these lower level parts independently of the Autopilot part, or even use them to build his own autopilot, keeping the bundle but not using any of the CAMP Autopilot parts. Alternatively, he may use only a subset of the part's facilities, supplying other required facilities with his own packages. These methods of use address the reuse techniques identified by Standish (Reference 14, p 496):

- Direct reuse of concrete modules [= high level reuse]
- Reuse after refinement [= lower level reuse]
- Reuse after modification [= independent reuse]

Actual use of CAMP parts has proven the effectiveness of the semi-abstract method. Most of the parts are themselves constructed from other parts; this is illustrated by the background bundling of Signal_Processing or Basic_Data_Types in the Autopilot bundle. Also, applications using CAMP parts have, in some cases, found that the higher level part is not complete for some special operations. In these situations, the CAMP users access the bundle, taking as much from the high level part as possible and building the rest from lower level entities in the bundle. The CAMP Kalman filter bundle, for example, contains a General_Vector_Matrix_Algebra part (see Figure 60.) This part is used extensively in the instantiation of CAMP Kalman filter parts. A user of the CAMP parts, needing additional functions not built into them, can build the required functions out of the General_Vector_Matrix_Algebra parts or develop his own operators to perform the same functions. Such was the case in the 11th Missile Application, where special-purpose matrix operations were required to meet performance constraints (see Volume II, Section III). By building these special operations with interfaces conforming to the CAMP General_Vector_Matrix_Algebra parts, the 11th Missile development team was able to use the high-level Kalman filter parts without modification.

SECTION VII

ADA COMPILER VALIDATION AND SOFTWARE REUSABILITY

1. INTRODUCTION

An important part of the CAMP project was the development of a part design methodology by which Ada parts can simultaneously be reusable, transportable, flexible, efficient, easy to use, and protected against misuse. These seemingly conflicting design goals are achieved by exploiting many of the advanced features of Ada, such as derived types and subprograms, generic units with default formal parameters, and subprogram overloading. Section VI discussed these features as they apply to design of the CAMP parts. This section discusses the impact of these features on parts implementation and compiler selection.

2. DISCUSSION

To achieve these design goals, the CAMP design method included the use of generic units, strong data typing, and generic object and subprogram parameter defaults.

- **Generic units:** The primary facility Ada provides which promotes reusability is the generic unit. Although some people in the Ada community have expressed a "fear" of this feature, MDAC-STL has embraced it wholeheartedly. Without generic units, reusability in Ada would not be achievable at a meaningful level. However, there is a risk associated with using generic units — Ada compilers must be able to implement them efficiently and correctly.
- **Strong data typing:** Among the most important capabilities in Ada is the ability to strongly type data. However, strong data typing has two characteristics which unnecessarily cause many people (including some part developers) to avoid it:
 - The use of strong data typing makes the design of generic packages and subprograms more complex.
 - The interaction of Ada typing rules with other Ada features such as generic units is non-trivial to master.

For these reasons, some software developers have developed Ada parts in a typeless fashion. We believe this is a mistake. Parts which are *typeless* are very prone to misuse. It is only reasonable that if the parts being developed are intended for long-term use, then it should be worth the effort to build them in the most protected fashion.

- **Generic object and subprogram parameter defaults:** As previously mentioned, the use of strong data typing causes generic units to be more complex. Specifically, the generic packages and subprograms must now import many operations and functions which would otherwise be visible to them implicitly through the scoping rules of Ada. If this drawback could not be overcome, it would

be a good argument against strong data typing. However, Ada provides a feature — the specification of defaults for generic object and subprogram parameters — which negates the drawback while still retaining the advantages. This feature, the generic unit, and its use for part design is further discussed in Section VI.3.a.

a. A Sample System

The complexity of reusable generic parts can range from extreme simplicity (see Figure 61) to considerable complexity (see Figures 62 and 63), with most falling somewhere in between (see Figure 64).

```
with CALENDAR;
generic
package Clock_Handler is

    function Current_Time return STANDARD.DURATION;

    function Converted_Time (Clock_Time : in CALENDAR.TIME)
        return STANDARD.DURATION;

    procedure Reset_Clock;

    procedure Synchronize_Clock
        (New_Time : in STANDARD.DURATION;
         Clock_Time : in CALENDAR.TIME := CALENDAR.CLOCK);

    function Elapsed_Time return STANDARD.DURATION;

end Clock_Handler;
```

Figure 61. Generic Units Can Be Very Simple

While most generic units have minimal complexity in and of themselves, their use in the development of a system can become quite involved. This is because even though an individual generic unit may be relatively independent of other generic units, it has probably been designed to be used in conjunction with other generic units.

Figure 65 illustrates the parts that may be required in the design of a small portion of a navigation system. In order to instantiate three north-pointing navigation parts (Coriolis_Acceleration, Radius_of_Curvature, and Latitude_Integration) using strong data typing where all floating point types are separate Ada data types, the following must occur:

1. Ten packages must be compiled into the user's library. The user himself requires six of these (indicated by the arrows going into the user application). These six require an additional four.

```

generic
  type Left_Indices    is (<>);
  type Right_Indices   is (<>);
  type Result_Indices  is (<>);
  type Left_Elements   is private;
  type Right_Elements  is private;
  type Result_Elements is private;
  type Left_Vectors    is private;
  type Right_Vectors   is private;
  type Result_Vectors  is private;
  X1 : in Left_Indices  := Left_Indices'FIRST;
  Y1 : in Left_Indices  := Left_Indices'SUCC(Left_Indices'FIRST);
  E1 : in Left_Indices  := Left_Indices'LAST;
  X2 : in Right_Indices := Right_Indices'FIRST;
  Y2 : in Right_Indices := Right_Indices'SUCC(Right_Indices'FIRST);
  E2 : in Right_Indices := Right_Indices'LAST;
  X3 : in Result_Indices := Result_Indices'FIRST;
  Y3 : in Result_Indices := Result_Indices'SUCC(Result_Indices'FIRST);
  E3 : in Result_Indices := Result_Indices'LAST;
  with function "+" (Left : Result_Elements;
                     Right : Result_Elements) return Result_Elements is <>;
  with function "-" (Left : Result_Elements;
                     Right : Result_Elements) return Result_Elements is <>;
  with function "-" (Right : Result_Elements) return Result_Elements is <>;
  with function "*" (Left : Left_Elements;
                     Right : Right_Elements) return Result_Elements is <>;
  with function Retrieved_Element
    (Vector : Left_Vectors;
     Index  : Left_Indices) return Left_Elements is <>;
  with function Retrieved_Element
    (Vector : Right_Vectors;
     Index  : Right_Indices) return Right_Elements is <>;
  with procedure Set_Element
    (Index : in    Result_Indices;
     Value : in    Result_Elements;
     Vector : out Result_Vectors) is <>;
function Generic_Cross_Product (Left : Left_Vectors;
                                Right : Right_Vectors)
  return Result_Vectors;

```

Figure 62. Some Generic Units Can Be Very Complex

2. The user must do the following before instantiating the navigation parts:

- Instantiate four versions of the square root package (GPMath.Square_Root) using data types and operators supplied by the basic data types (BDT) package.
- Instantiate four versions of the vector operations package (CVMA.Vector_Opns) using data types and operators supplied by BDT and the square root functions contained in the packages previously instantiated by the user.
- Instantiate a cross product function using scalar data types and operations supplied by BDT, along with vector data types and operations obtained from three separate instantiations of CVMA.Vector_Opns.

3. The three navigation parts can then be instantiated using:

- Scalar data types and operators supplied by BDT.

```

package Direction_Cosine_Matrix_Operations is

  generic
    type Earth_Axes      is (<>);
    type Navigation_Axes is (<>);
    type Sin_Cos_Ratio   is digits <>;
    type Real            is digits <>;
    with function Sqrt (Value : Sin_Cos_Ratio) return Sin_Cos_Ratio is <>;
    with function "*" (Left  : Sin_Cos_Ratio;
                      Right : Sin_Cos_Ratio) return Real is <>;
    with function "*" (Left  : Sin_Cos_Ratio;
                      Right : Real) return Sin_Cos_Ratio is <>;
    Greenw : in Earth_Axes := Earth_Axes 'FIRST;
    Right   : in Earth_Axes := Earth_Axes 'SUCC(Earth_Axes'FIRST);
    Polar   : in Earth_Axes := Earth_Axes 'LAST;
    East    : in Navigation_Axes := Navigation_Axes 'FIRST;
    North   : in Navigation_Axes := Navigation_Axes 'SUCC(Navigation_Axes'FIRST);
    Up      : in Navigation_Axes := Navigation_Axes 'LAST;
  package CNE_Operations is

    type CNE_Matrices is array (Earth_Axes, Navigation_Axes) of Sin_Cos_Ratio;

    function CNE_Initialized_From_Reference (Ref_CNE_2_1 : Sin_Cos_Ratio;
                                             Ref_CNE_2_2 : Sin_Cos_Ratio;
                                             Ref_CNE_3_1 : Sin_Cos_Ratio;
                                             Ref_CNE_3_2 : Sin_Cos_Ratio;
                                             Sign_of_2_3 : INTEGER;
                                             Sign_of_3_3 : INTEGER)
      return CNE_Matrices;

  generic
    type Earth_Positions is digits <>;
    type Angles          is digits <>;
    with procedure Sin_Cos (Input   : in Angles;
                          Sin_Value : out Sin_Cos_Ratio;
                          Cos_Value : out Sin_Cos_Ratio) is <>;
    with procedure Sin_Cos (Input   : in Earth_Positions;
                          Sin_Value : out Sin_Cos_Ratio;
                          Cos_Value : out Sin_Cos_Ratio) is <>;
    function CNE_Initialized_from_Earth_Reference
      (Wander_Angle : Angles;
       Latitude     : Earth_Positions;
       Longitude    : Earth_Positions) return CNE_Matrices;

  end CNE_Operations;

end Direction_Cosine_Matrix_Operations;

```

Figure 63. Nested Generic Units Can Be Very Complex

- Scalar data types and trigonometric functions supplied by an instantiation of the standard trig package contained in BDT (BDT.Trig).
- Vector types and operations supplied by the four instantiations of CVMA.Vector_Opns.
- Data constants supplied by the WGS72 ellipsoid metric data package (WGS72) and the WGS72 ellipsoid unitless data package (WGS72U).
- User-defined data types and objects.

```

generic
  type Unit_Vectors is private;
  type Sin_Cos_Ratio is digits <>;
  with function "/" (Left : Unit_Vectors;
                    Right : Sin_Cos_Ratio) return Unit_Vectors is <>;
  with function Cross_Product (Left : Unit_Vectors;
                              Right : Unit_Vectors)
    return Unit_Vectors is <>;
  with function Vector_Length (Input : Unit_Vectors)
    return Sin_Cos_Ratio is <>;
function Unit_Normal_Vector
  (Unit_Radial_A : Unit_Vectors;
   Unit_Radial_B : Unit_Vectors) return Unit_Vectors;

```

Figure 64. Most Generic Units Have Minimal Complexity

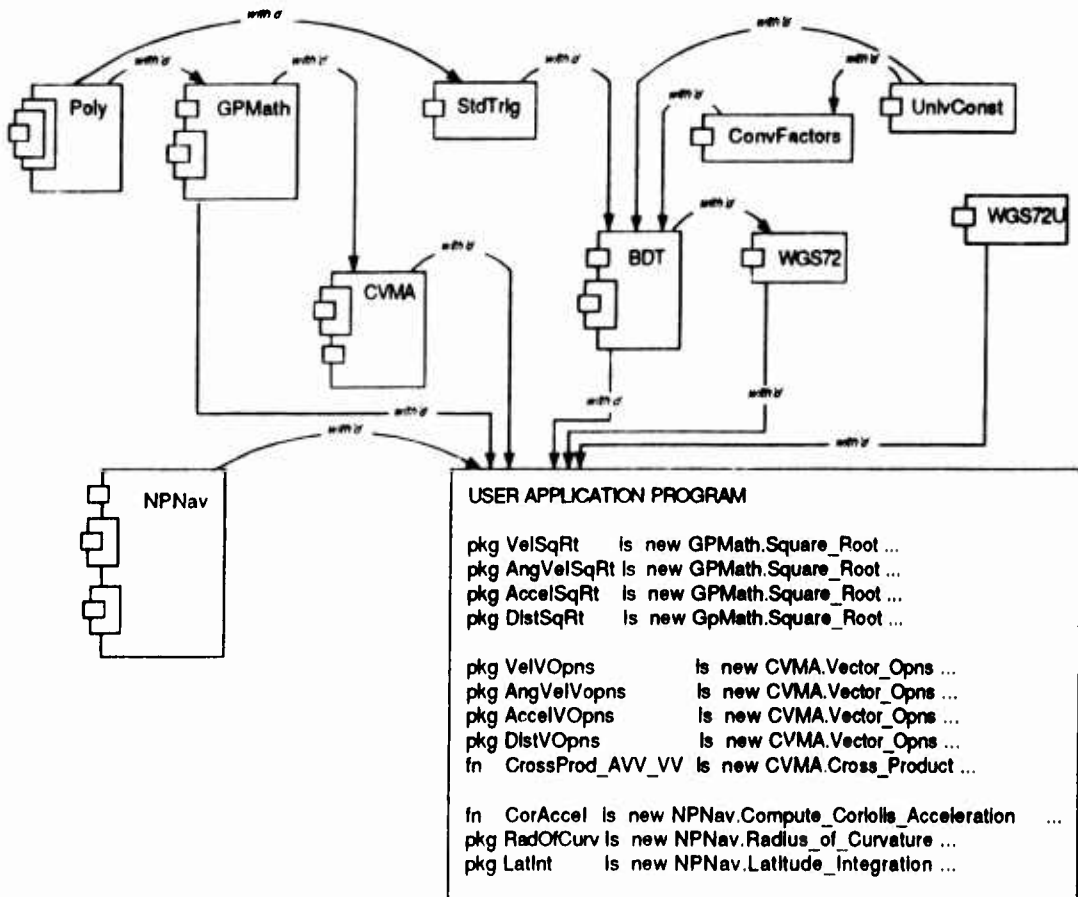


Figure 65. Assembling a North-Pointing Navigation System

b. CAMP Experience With Ada Compilers

The development and use of truly good, flexible, reusable software will succeed only if generic units are fully supported by Ada compilers. Yet, during the CAMP project, we observed that *validated Ada compilers frequently cannot handle any but the simplest generic units*.

During the CAMP project, there were many opportunities to see how compilers handled generic units. Three compilers were used on the CAMP project (two validated and one prevalidated, of which two were 1750A-targeted), and versions of the CAMP parts were submitted to three additional validated compilers. Of these six Ada compilers, only one validated compiler was able to handle the parts submitted to it, and even that one was able to do so only after a year and a half of the CAMP team working with the vendor.

While various problems were encountered, they all had one thing in common — they involved the use of generic units. Some of these problems are enumerated below.

- Difficulties in handling a multitude of instantiations. Using the code represented in Figure 65 as an example, one compiler was able to compile all of the CAMP parts required to develop the user application. However, when an attempt was made to compile the user code, the compiler crashed. (It should be noted that the user code was legal Ada and did compile on another validated compiler.)
- Difficulties in declaring derived real types when the base type was a generic formal type, particularly if a range constraint was added (see Figure 66). Attempting to do this sent one compiler into an infinite loop. Another compiler allowed the derived type to be declared, but encountered an internal error when an attempt was made to restrict the range of the newly declared derived type.
- Incorrect passing of the value of a generic actual object to a generic actual subprogram — the compiler sent a value of 0.0 regardless of the actual value of the object. The generic actual object was a named number defined in a package which had to be imported by the user application. This error occurred only when strong data typing was employed (i.e., a different generic actual type was specified for each of the generic formal data types), not occurring when FLOAT was used for all actual types. Additionally, even when strong data typing was employed, this error did not occur if an explicit type conversion was performed on the object at the time it was used in the instantiation and also did not occur if a literal was used instead of the named number.
- Inability to resolve overloading of operators when a generic formal subprogram ("+" in this case) matched an operator already defined by the language (see Figure 67).
- Inability to identify generic actual subprograms to be used as defaults even though they were directly visible. Two variations of this problem occurred and are illustrated in Figure 68. In the first, the correct subprogram was directly visible as the result of 'with' and 'use' clauses on the subprogram's package. In the second case, the correct subprogram was directly visible since it was a generic actual subprogram to the part where problems were encountered.

- An inability to handle separate compilation of generic units, even though compiler documentation indicated this optional feature was implemented.

This code sent one compiler into an infinite loop:

```
generic
  type Angles is digits <>;
  type Ratios is digits <>;
  Pi : in Angles;
package StdTrig is
  type Radians is new Angles;      *
end StdTrig;
```

* - This statement caused the problem

This code caused another compiler to encounter an internal error:

```
generic
  type Angles is digits <>;
  type Ratios is digits <>;
  Pi : in Angles;
package StdTrig is
  type Radians      is new Angles;
  type Sin_Cos_Ratio is new Ratios range -1.0..1.0;  #
end StdTrig;
```

- This statement caused the problem

Figure 66. Some Compilers Couldn't Handle Type Derivations

Specification:

```
generic
  type M_Indices      is (<>);
  type N_Indices      is (<>);
  type P_Indices      is (<>);
  type Left_Elements  is digits <>;
  type Right_Elements is digits <>;
  type Result_Elements is digits <>;
  type Left_Matrices  is array (M_Indices, N_Indices) of Left_Elements;
  type Right_Matrices is array (N_Indices, P_Indices) of Right_Elements;
  type Result_Matrices is array (M_Indices, P_Indices) of Result_Elements;
  with function "*" (Left : Left_Elements;
                    Right : Right_Elements) return Result_Elements is <>;
  with function "+" (Left : Result_Elements;
                    Right : Result_Elements) return Result_Elements is <>;
function Matrix_Matrix_Multiply
  (Left : Left_Matrices;
   Right : Right_Matrices) return Result_Matrices;
```

Body:

```
function Matrix_Matrix_Multiply
  (Left : Left_Matrices;
   Right : Right_Matrices) return Result_Matrices is
  Answer : Result_Matrices;
begin
  for M in M_Indices loop
    for P in P_Indices loop
      Answer(M,P) := 0.0;
      for N in N_Indices loop
        Answer(M,P) := Answer(M,P) +      #
          Left(M,N) * Right(N,P);
      end loop;
    end loop;
  end loop;
  return Answer;
end Matrix_Matrix_Multiply;
```

- Compiler was unable to resolve this overloading

NOTE: Constrained arrays were used in the design of this part in order to improve the efficiency of the part. While it was recognized that unconstrained arrays would have made the part more flexible and hence more reusable, the need for efficiency for real-time embedded applications was considered of greater importance.

Figure 67. Overloaded Operator Caused Problems for Compiler

When attempting to instantiate the following generic:

```
generic
  type Angles      is digits <>;
  type Inputs      is digits <>;
  type Outputs     is digits <>;
  type Sin_Cos_Ratio is digits <>;
  with function Sin (Input : Angles) return Sin_Cos_Ratio is <>;
  function Example (Input : Inputs) return Outputs;
```

One compiler couldn't resolve the default even though the appropriate subprogram was directly visible through 'with' and 'use' clauses:

```
generic
  type Angles is digits <>;
  type Ratios is digits <>;
  package StdTrig is
    type Radians is new Angles;
    type Sin_Cos_Ratio is new Ratios range -1.0..1.0;
    function Sin (Input : Radians) return Sin_Cos_Ratio;
  end StdTrig;

  with StdTrig;
  package BDT is
    type Real is digits 9;
    package Trig is new StdTrig (Angles => Real,
                                Ratios => Real);
  end BDT;

  with BDT; use BDT;
  with Example;
  procedure User_Application is

    use BDT.Trig;

    function Attempted_Instantiation is new Example
      (Angles      => BDT.Trig.Radians,
       Inputs      => BDT.Real,
       Outputs     => BDT.Real,
       Sin_Cos_Ratio => BDT.Trig.Sin_Cos_Ratio);

  begin
    ...
  end User_Application;
```

\ > problem encountered
/ with this
instantiation

Another compiler couldn't resolve the default even though it was visible as a generic formal subprogram:

```
generic
  type Angles      is digits <>;
  type Inputs      is digits <>;
  type Outputs     is digits <>;
  type Sin_Cos_Ratio is digits <>;
  with function Sin (Input : Angles) return Sin_Cos_Ratio is <>;
  package Sample is
    ...
  end Sample;

  with Example;
  package body Sample is

    function Attempted_Instantiation is new Example
      (Angles      => Angles,
       Inputs      => Inputs,
       Outputs     => Outputs,
       Sin_Cos_Ratio => Sin_Cos_Ratio);

  end Sample;
```

\ > problem encountered
/ with this
instantiation

Figure 68. Compilers Had Problems Finding Default Subprograms

c. Compiler Validation

The Ada Compiler Validation Capability (ACVC) test suite is designed to ensure a certain level of quality and confidence in Ada compilers, and to a large extent has succeeded. The CAMP experience, however, indicates that notable inadequacies exist in the area of generic units. These inadequacies could have a significant negative impact on the future development and use of reusable software.

The use of generic units is vital to the development of good reusable parts, yet we have found that it is one area where even validated compilers often are lacking. Based on the CAMP experience, most validated Ada compilers seem to be able to handle simple generic units, many are unable to handle complex generic constructs, and most are unable to handle the complex mix of generic units that is required to assemble a software system from a collection of reusable generic parts.

The tests in the ACVC test suite seem to be geared to test or demonstrate only a single objective. While this has ensured that validated compilers can generally handle simple generic units, it is probably why an Ada compiler can be validated though unable to handle complex generic units, and is certainly why a complex mix of generic units is beyond the ability of most validated Ada compilers. While this approach may have been appropriate in the beginning when there was a desire to get an initial set of validated compilers, we feel the time has come to modify this approach.

SECTION VIII

CONCLUSIONS AND RECOMMENDATIONS

Given the pathfinding nature of CAMP-2, it is not surprising that many lessons were learned concerning the use of Ada to develop reusable software for real-time, embedded (RTE) applications. One of the primary benefits of the CAMP-2 project has been in sharing these "lessons learned" with the DoD software engineering community. This section discusses the major conclusions reached during the CAMP-2 project and presents recommendations based on these conclusions.

1. ON THE APPROPRIATENESS OF ADA FOR REUSABLE SOFTWARE

A primary design goal of the Ada programming language was to promote reuse of software. The designers of Ada addressed this goal in two ways. First, Ada was designed to facilitate transporting applications between different computer architectures. Second, Ada was designed to facilitate the development of code units which could be transported between different applications.

Conclusion #1

With a few minor exceptions, Ada achieves its reusability design goal.

Conclusion #1 is substantiated by two facts. First, many Ada applications have been transported between different computer architectures at a small fraction of the cost traditionally associated with rehosting non-Ada applications. Second, Ada parts are rapidly becoming available from a variety of sources (including CAMP) and these parts are being reused. The CAMP parts have been distributed to over 120 DoD agencies and contractors who are exploring their utility in a wide spectrum of applications (e.g., avionics, ballistic missiles, space station control, etc.). McDonnell Douglas is in the process of using the CAMP parts on a number of applications.

There are several primary factors which have led to Ada's success in the area of reusability.

- The DoD has rigidly adhered to a standard language definition
- Ada's package feature provides the user with the means to encapsulate machine and application dependencies
- Ada's generic unit feature provides the ability to broaden the domain applicability of reusable components
- Ada allows the underlying machine architecture to be hidden

However, there are some aspects of Ada which need to be improved from the perspective of reusability. Section V describes the rationale for these recommendations in more detail.

Recommendation #1

The definition of Ada should be changed to allow address objects to be passed as generic parameters.

Recommendation #1 will promote reuse of machine control and communication software. For example, during the CAMP-2 11th Missile Demonstration, a Bus Interface Module component was developed which could be reused between the Guidance computer TLCSC and the Navigation computer TLCSC. The only difference was the actual physical address of the bus discretes. Since address objects cannot be generic parameters, manual changes had to be made to the component in order reuse it.

Recommendation #2

The definition of Ada should be changed to allow representation clauses to be defined within a package body.

Recommendation #2 will uncouple the physical and logical definitions of Ada entities and hence promote reuse. The current requirement to define a representation clause within the same declarative scope as the entity declaration, means that if a different application wants to reuse a component with the same logical representation but a different physical representation, it must manually change the component.

Recommendation #3

The definition of Ada should be changed to allow a single, unmodified Ada specification to be used with multiple bodies within a single application.

Recommendation #3 will increase the degree to which Ada specifications can be reused without manual modifications. For example, currently, to use two different bodies to implement a single abstract data structure within an application, the specification must be copied and manual name changes must be made to it.

Recommendation #4

The definition of Ada should be changed to require a compiler to support separate compilation of generic units and subunits.

Recommendation #4 will decrease compilation overhead when software components are reused. This change will have a significant beneficial impact on reuse since there are real advantages to separate compilation in the areas of configuration management, project management, and compilation time.

Recommendation #5

The definition of Ada should be changed to allow procedural data types.

Recommendation #5 will promote reuse within applications that require dynamic reconfiguration and within Artificial Intelligence applications.

2. ON THE APPROPRIATENESS OF ADA FOR REAL-TIME EMBEDDED REUSABLE SOFTWARE

Another Ada design goal was that it be suitable for use in RTE applications. This implies that Ada must not only provide efficient higher-order language (HOL) features, but must also allow the programmer, when needed, to have direct control over the representation of Ada entities, access the computer hardware directly, trade-off space and execution time, closely control and be able to characterize the dynamic behavior of a program, and in general, to perform operations which a non-RTE programmer might consider "unsafe".

The appropriateness of Ada for RTE applications depends on four factors. These factors are discussed in the following subsections.

- Is Ada an effective language for RTE applications?
- Are there any features in Ada which must be used in RTE applications but are inherently inefficient?
- Are Ada compilers sufficiently effective for RTE applications?
- Is the code produced by Ada compilers sufficiently efficient for RTE applications?

Obviously, when one addresses either of the last two issues, it must be done based on experience with a particular set of compilers during a specific period of time. Thus, the conclusions reached on the CAMP-2 project concerning Ada compilers are dependent upon the specific compilers used and the time period in which they were used.

a. On the Effectiveness of Ada

A determination of the effectiveness of the Ada language for RTE applications is essentially a determination as to whether all the functional requirements of RTE applications can be achieved within the language. In other words, if there are operations that an RTE application typically needs to perform, and cannot do so using the language, then Ada would be judged ineffective to some degree.

Conclusion #2

Ada is an effective language for real-time embedded applications.

Conclusion #2 is based on the CAMP-2 11th Missile Application experience. The 11th Missile Application was constructed using only 21 assembly language statements; this equates to 0.1% of the total software (see Volume II for more details). With the exception of two small functions, all the functional requirements of the 11th Missile Application were achieved using Ada. In fact, the 21 assembly language statements could have been coded using Ada's machine code insertion feature. In the case of the 11th Missile Application, the functions which required the use of assembly language had to do with operating system idiosyncracies. But, every RTE system tends to have its own idiosyncracies. The reassuring fact is that Ada can handle all these situations, assuming that machine code insertion is supported. In existing RTE applications that use HOLs, the percentage of assembly language used for functional reasons² is usually much higher than that experienced on CAMP.

A common myth concerning Ada, which needs to be dispelled, is that Ada stops a programmer from doing certain operations which are considered to be "unsafe" but which RTE programmers need to do. If this myth were true, it would indeed be a major problem with Ada. The reality is that the software engineering discipline has recognized that certain programming paradigms are dangerous (i.e., their use frequently leads to errors) and in most cases these paradigms can be avoided. In some languages, like Pascal, a dogmatic approach has been adopted and these paradigms are outlawed completely. This is not the case with Ada. Ada tries to balance the goal of promoting sound software engineering principles and the reality that upon occasion a programmer needs to do something that is dangerous. Thus, Ada allows the programmer to use "dangerous" paradigms, but doesn't make their use too easy — a suitable compromise in the authors' opinions. An example of an operation which is often considered dangerous but which is essential in an RTE application is overlaying two data structures on the same data.

Conclusion #3

A full implementation of the Chapter 13 features of Ada is essential in real-time, embedded applications.

Conclusion #3 highlights the fact that the effectiveness of Ada for RTE applications is highly dependent upon the extensive use of Ada features which are defined in the Language Reference Manual as optional. These features are popularly called the "Chapter 13 features" of Ada. Projects must be sensitive to the fact that in RTE applications, the Chapter 13 features of Ada should not be considered optional.

²In addition to using assembly language for functional reasons, RTE applications frequently have to replace HOL code with assembly code for performance reasons.

b. On the Inherent Efficiency of Ada

There has been a great deal of debate within the DoD software engineering community concerning the efficiency of the Ada language. Much of this debate has concentrated on the efficiency of Ada features, such as tasking, exception handling, and generic units, which are not in our traditional RTE languages.

Conclusion #4

There appears to be no Ada features which are inherently inefficient.

While it is true that the efficiency of the advanced Ada features as implemented by the current generation of Ada compilers leaves something to be desired, a preliminary analysis indicates that there is nothing within the definition of the Ada language which requires them to be inefficiently implemented. The inefficient results mainly from a lack of global optimization in most of the current Ada compilers.

Conclusion #5

There are Ada features which require a global optimizer to be sufficiently efficient for severely constrained RTE applications.

For example, consider Ada generic units. When a generic is compiled, the compiler is unaware of the values of the generic parameters and must, therefore, generate code which can handle any situation. This results in code that will be relatively inefficient. However, if a compiler had a sufficiently powerful global optimizer, it could use the information known at the point(s) of instantiation and optimize the code for the generic unit.

In every situation where inefficiencies were encountered on CAMP, we were able to determine that a sufficiently powerful optimizer could have corrected the situation. Unfortunately, few, if any, of the current generation of Ada compilers implement optimizers which are sufficiently powerful for severely constrained RTE applications. The next two subsections discuss Ada compiler issues in more detail.

c. On the Effectiveness of Ada Compilers

Given that the Ada language is effective for reusable RTE software, a determination of the effectiveness of Ada compilers for the same type of software is based on two factors. First, the compiler must properly handle all mandatory Ada features. The ability to properly handle Ada generic units is of special importance given the crucial role that generic units play in reusability. Second, the compiler must handle all Chapter 13 features in Ada. As previously discussed, in RTE applications these features are essential.

Conclusion #6

Ada compilers do exist which are effective for real-time embedded application.

Conclusion #6 is based on the fact that the CAMP 11th Missile Application, a true RTE system,

was implemented using only 21 assembly language statements. This system was tested in a hardware-in-the-loop simulation environment on a 1750A processor. The particular 1750A Ada compiler used for this demonstration had an excellent implementation of the Chapter 13 features of Ada. This is not to imply that these compilers handle all Ada language constructs efficiently. For example, even the 1750A Ada compiler used for the 11th Missile demonstration had difficulties with complex generics and efficient throughput for tasking.

Recommendation #6

The DoD needs to enhance its Ada Validation process.

Too many validated compilers have detectable errors. Recommendation #6 is based on the fact that many DoD project managers mistakenly believe that if the DoD says a compiler is validated, then it must be OK to use. It is important to note that it was only in the final months of the CAMP-2 project that we had a 1750A Ada compiler that met most of our RTE effectiveness requirements (the exception was in the area of generic units). We spent a significant amount of time and effort testing compilers, reporting problems, and working with the compiler developers to correct the problems. During a large portion of this time the compilers were validated.

Recommendation #7

During the next few years, DoD mission-critical real-time embedded Ada projects should establish a contractual relationship with their compiler developer to reduce risk.

Until Ada compilers are fully mature, critical RTE Ada projects will be well served to acquire the highest level of maintenance support from their compiler developer or to put them under a special contract. If problems are found with the compiler, it is unrealistic to expect major projects to wait for the next scheduled release to get the problems fixed. On CAMP-2, it was mutually advantageous for McDonnell Douglas and the selected 1750A Ada compiler supplier to work closely together.

Conclusion #7

Ada compilers do exist which are effective for applications that want to use reusable software components.

Conclusion #7 is based on the fact that there exist compilers which handle Ada generic units effectively. The CAMP project has had a great deal of success with the DEC VAX Ada compiler.

Conclusion #8

CAMP data indicates that the current generation of Ada/1750A compilers do not support generic units well and this lack of support will hinder real-time embedded applications that want to use reusable software components.

Conclusion #8 is a disappointing result based on the present immaturity of 1750A Ada compilers. Obviously, we can only extrapolate the situation with 1750A Ada compilers to other RTE compilers. The basic problem is that many validated Ada compilers, especially those which target RTE

computers, do not handle generic units correctly. Most validated compilers handle simple generic units in an adequate fashion, but a great majority of Ada compilers will have problems with non-trivial generic units. Section VII discusses the types of situations which cause most compilers to have problems.

With the particular 1750A Ada compiler used on the 11th Missile Application (which we believe is one of the best of its type), we spent a significant amount of time and effort working with the compiler developer to overcome problems associated with generic units. Even after all this effort, the result was that all the CAMP generic parts compiled, most of them linked, but many of them caused abnormal program execution due to compiler errors. To overcome these compiler problems, we had to manually instantiate approximately 42% of the CAMP parts used on the 11th Missile Application.

Recommendation #8

The Ada Validation suite must be changed to incorporate tougher tests on generic units.

During CAMP-2, a benchmark was developed which rigorously tests a compiler's ability to deal with non-trivial Ada generic units (see Volume III) . A test based on this benchmark would give Ada compiler developers the incentive to effectively handle generics — if they don't, they would lose their validated status.

d. On the Efficiency of Ada Compilers

While the efficiency of the code produced by Ada compilers is important to all types of applications, it is critical for RTE applications. The performance requirements of RTE applications are typically non-negotiable. The RTE software engineer cannot trade-off run-time speed for a more maintainable software system, nor can she arbitrarily accept a larger object code size for the sake of reusability.

Another aspect of efficiency which is important to RTE applications and which many non-RTE software engineers often fail to understand is that of micro-level efficiency. In other words, in addition to being concerned with macro-level efficiency issues such as the selection of appropriate algorithms, the RTE software engineer is often concerned with the efficiency of specific language constructs. The authors have had frequent conversations with other researchers in the area of reusability in which the other researchers could not understand why the CAMP parts were designed as semi-abstract parts as opposed to being developed as pure abstract data structures. In their value system, the benefits of pure abstract parts more than accounted for a "few more assembly language statements." However, in many RTE applications, such as missile guidance and navigation systems, a few more statements in a high rate (e.g., 100 hertz) task can make the difference between an effective weapon system and one that doesn't achieve its operational requirements.

Conclusion #9

CAMP data indicates that current implementations of Ada tasking are sufficiently inefficient to cause concern in severely constrained RTE applications.

The speed of an Ada task rendezvous on most compilers is such that a RTE programmer should

avoid its use for any fast loops or high rate interrupts. Some RTE Ada compiler developers have recognized this problem and provided an alternative method of handling interrupts.

Conclusion #10

CAMP data indicates that current implementations of Ada generics are sufficiently inefficient to cause concern in severely constrained RTE applications.

Currently, there are two approaches used by Ada compiler developers to implement generics: the *single body* approach and the *multiple body* approach. With the single body approach, a single unit of code is generated that can handle any type of instantiation; this approach trades speed for a smaller code size. With the multiple body approach, a separate set of code is generated for each instantiation; this approach trades code size for better speed. In general, we believe that the multiple-body approach is better. Our preference is based on the observation that most parts are instantiated only once within an application. Thus, using the multiple body approach provides both a speed and storage advantage. However, both approaches suffer from the inability of most compilers to perform global optimization.

Recommendation #9

Ada compilers should be able to alternate between single body and multiple body generic implementation based on either implicit or explicit information.

In the best case, the compiler should be able to use both the single body and the multiple body implementation of generic units. Ideally, the compiler would make the choice of the implementation mechanism based on data provided by pragmas and/or a global optimization analysis.

Conclusion #11

CAMP data indicates that current implementations of Ada exceptions are sufficiently inefficient to cause concern in severely constrained RTE applications.

Because of the semantics of Ada exceptions, some Ada compilers generate code which waste a significant amount of storage. In effect, they keep extra copies of data until it can be verified whether or not an exception has been raised. In many cases this extra storage is not significant, but in some cases where the data being duplicated is extensive, e.g., an Kalman filter arrays, this method can cause severe memory problems.

Conclusion #12

With the exception of the inefficiencies due to generic units, tasking, and exception handling, current Ada compilers appear to have efficiency equivalent to other HOL compilers used in RTE applications.

When one ignores the advanced features of Ada, computational-intensive benchmarks show that Ada compilers perform as well as JOVIAL compilers.

Conclusion #13

The ability of Ada compilers to perform global optimizations is critical to the successful use of Ada and the reuse of Ada parts in RTE applications.

If there is one major message concerning compiler efficiency that was quite clear on CAMP, it is that Ada compilers need a global optimizer to be sufficiently efficient for RTE applications, with or without reuse. This need is driven by several factors.

- Ada's features promote design of highly modularized software, thus, Ada software is usually implemented by means of a high number of small units. If an Ada compiler cannot optimize across unit boundaries, a large amount of potential optimization will be lost.
- Reusable parts and data are typically bundled together into cohesive packages to make their use and maintenance easier. If a compiler cannot perform global analysis to identify and eliminate *dead code* and *dead data*, some of the benefits of reusable parts will be lost when the user has to manually eliminate these items.
- The Ada generic unit is an extremely powerful concept, but to make use of it on RTE applications, the compiler must be able to optimize the code generated based on the context of the instantiation.
- Like generic units, Ada's exception handling features are very useful, but compilers must not penalize the user who has decided not to use the features.

Given that few, if any, Ada compilers currently implement a sufficiently powerful global optimizer for RTE applications, an important question is whether an application can avoid inefficiencies by avoiding certain Ada features. The answer is not always. Certainly an application can avoid generics and hence avoid the overhead of a generic. However, this is not the case with exceptions. Whether or not an application uses exceptions, it will pay the costs associated with detecting and communicating exceptions because without a global optimizer, the compiler cannot know that an exception handler is not declared at a higher level.

3. ON THE DEVELOPMENT OF THE CAMP PARTS

During CAMP-2, McDonnell Douglas developed 454 parts consisting of over 16,000 lines of operational Ada code and another 27,000 lines of Ada test code. From this work, we developed a number of conclusions concerning the use of Ada and the development of parts.

Conclusion #14

The use of Ada results in improved development productivity.

MDAC-STL carefully collected data concerning the effort expended and the resulting size of the CAMP parts. This data shows that overall productivity for developing the CAMP parts was 258 LOC/MM. One software cost estimating model, COCOMO, estimated productivity at 160 LOC/MM. Section II of this volume describes the productivity analysis for the CAMP parts development in greater detail. We attribute this increased productivity to four factors.

- The use of Ada
- The use of good people
- The use of good tools
- The reuse of software

Few people have doubted that Ada would increase the productivity of the software maintenance process, but one of the unresolved questions within the Ada software engineering community has been whether the use of Ada would help developmental productivity on the first set of projects on which it was used. We believe the use of Ada was the primary factor behind our higher than expected productivity on the CAMP parts development task. In addition to providing a complete set of structured control constructs and a highly readable language, the Ada package featured allowed us to identify clear interfaces between the different people working on the parts, and hence, promoted a high degree of parallelism in the parts development.

Conclusion #15

Ada's support for programming-in-the-large is one of its chief advantages from a management perspective.

It is worthwhile noting that one important reason that the use of Ada was a benefit to our developmental productivity was that we had an excellent compiler to develop the parts — the DEC VAX Ada compiler. If one had to struggle with an immature compiler, productivity would be severely decreased.

Conclusion #16

The use of strongly typed software parts has significant benefits to the parts user, but complicates the development of parts.

One of the primary decisions the CAMP team had to make very early in the development of the CAMP parts was how extensively to use data typing. The chief advantage of making the parts strongly typed was the high degree of protection against misuse of the parts such typing would provide. The disadvantage of using strong typing was the increased complexity of developing the parts. The interactions between types and generics are much more complex than they appear to a casual user of Ada.

Initially, we had some doubts about the use of strong typing. Was it worth the extra effort to avoid data typing errors? We surveyed some of our on-going missile projects and asked them if data typing errors were a problem. Somewhat to our surprise, we found that the misuse of data was considered to be a significant problem area. Given the large number of different types of data used in a missile applications, programmers sometimes made "stupid" mistakes (e.g., mixing radians and degrees) and these types of errors were frequently not detected until the software was tested; at this point they were very difficult to isolate. Based on this information, we decided to use strong typing in the development of the CAMP parts. After all, the parts would be developed once, but used many times.

Conclusion #17

It costs more to develop reusable parts than to develop customized software.

While Conclusion #17 is hard to quantify, it is our observation that, depending on the experience of the part developer, it costs about 5% to 10% more to develop good reusable parts than it costs to develop a customized unit of software. The part developer has to not only meet the functional requirements of a specific application, he also has to think about how to make the part general enough for a set of applications without losing a significant degree of efficiency.

Recommendation #10

Parts should be developed by a parts development team driven by project needs.

We envision three ways in which parts could be developed.

- By projects
- By an independent parts development group
- By a project-directed parts development group

The problem with the first approach is that few projects have the extra resources to make good parts. The typical DoD software project has a short schedule and a tight budget, and few project managers will divert their people from their primary task of meeting the contract requirements. Additionally, the first approach does not allow an organization to develop a cadre of parts development expertise which will result in lower parts development costs. The problem with the second approach is that, over time, such a group tends to lose touch with projects' needs and will eventually start producing parts that no one wants. The third approach is based on projects providing the parts developers with draft parts and part needs. This approach is the one we prefer. It allows an organization to develop a cadre of expert part developers but provides direction for them from the projects.

Conclusion #18

Software parts for RTE applications must be developed to be semi-abstract.

The developer of reusable parts for real-time, embedded applications must be sensitive to the fact that frequently the conceptual elegance of a part has to be sacrificed to obtain the required degree of efficiency. While academicians might insist that all parts be developed as pure abstract objects (i.e., the internal structure is hidden from the user), the realities of RTE applications frequently demand that a user access the internal structure of a part. Fortunately, the choice is not between an abstract part and a non-abstract part. A design approach exists, which we refer to as *semi-abstraction*, in which a part provides the user with both an abstract interface and a mechanism for directly accessing the internal structure. Section VI of this volume discusses this issue in greater detail.

4. ON THE BENEFITS OF USING PARTS

Conclusion #19

The use of Ada software parts can increase productivity.

MDAC-STL collected data on the effort expended and the resulting size of the 11th Missile Application. This data shows that overall productivity was 419 LOC/MM, and indicates that productivity can be increased by up to 15% by using the CAMP parts.

Productivity on the 11th Missile Application was lowered by difficulties with Ada/1750A compilers. Separate statistics on the amount of time spent trouble-shooting the selected compiler were not kept, so it is impossible to tell precisely the effect on productivity. However, we do know that, of 153 software errors found during testing, 96 were compiler errors and 57 were errors in CAMP-developed code. It seems reasonable, therefore, to assume that half the testing time was spent debugging the compiler. Incorporating this assumption, the productivity of the 11th Missile development would rise to 572 LOC/MM.

Section III of Volume II describes the productivity analysis for the 11th Missile Application in greater detail.

5. ON THE COST-EFFECTIVENESS OF CAPTURING SCHEMATIC COMMONALITY

Conclusion #20

Some important types of commonality cannot be captured in Ada.

Early in the CAMP program, we realized that there were types of commonality that existed within most domains that either could not be captured using Ada alone, or could not be captured efficiently using Ada alone. We refer to this type of commonality as *schematic commonality*. To capture this type of commonality requires a tool which can build Ada code when given the requirements of a particular application. We refer to these tools as *schematic component constructors*; several of these constructors were built and used on CAMP. Section IV describes this work in more detail.

Conclusion #21

Schematic Component Constructors have high value.

As an example of the utility of a schematic component constructor, consider the case of the CAMP Kalman Filter Constructor. A novice user can specify his requirements for a new Kalman filter in about two minutes using this constructor. It takes the constructor about another minute to generate the Ada code. In a typical situation, the Kalman Filter Constructor will generate 387 Ada LOC and use another 1553 CAMP parts LOC. The bottom line is that the user gets 1940 LOC for three minutes of work.

Based on the number of lines of code generated by the Kalman Filter Constructor for the 11th Missile Application, we estimate that a 28% productivity improvement could be obtained just from using the Kalman Filter Constructor.

Recommendation #11

More research needs to be performed to develop an approach for building schematic component constructors.

Although we believe that the utility of schematic component constructors is high, the current approach to their construction requires a large development effort and the resulting tool is not easily modified. One potential solution to these problems is to develop a constructor-constructor, i.e., a tool that would be capable of generating a wide variety of schematic component constructors. One approach to such a constructor-constructor would involve the use of an interactive Ada pre-processor.

6. ON THE CATALOGING OF PARTS

During CAMP-2, MDAC-STL built a prototype Ada parts catalog. We drew two major conclusions from this work.

Conclusion #22

Cataloged Ada parts should be classified by logical operations, not physical Ada units.

The CAMP parts catalog was implemented so that the basic units being cataloged were Ada units. Upon reflection, and after having used this catalog, we believe this approach has two significant disadvantages.

- When viewing parts, the user gets entire Ada units and then has to locate the portions of interest; this is less than optimal.
- Too many entities are cataloged under the current scheme. This can lead to user frustration and result in the parts not being used.

We believe that a better approach would have been to catalog the logical parts, not the physical Ada code units. For example, the catalog should tell the user that it has an entry for a unbounded LIFO queue, not that it has a package specification called LIFO_QUE and a package body with the same name. Using this paradigm, the user would search for logical parts and then, if needed, the user could examine the Ada structure of these parts.

Conclusion #23

The taxonomy(ies) used by an Ada parts catalog should be soft-coded.

A software parts taxonomy³ is an important component of every software parts catalog. One of the lessons learned on CAMP-2 was that regardless of the time and effort spent in developing the taxonomy⁴, the taxonomy will change over time. No one can foresee all possible classes of parts. Likewise, the distinc-

³A mechanism for classifying software parts

⁴The categories into which parts are classified

tion of taxa is an extremely subjective activity. Given these factors, we recommend that software parts libraries "soft-code" their taxonomies so that they can naturally evolve over time.

APPENDIX A

PARTS DATA BASE

1. INTRODUCTION AND BACKGROUND

During development of the CAMP parts, certain information about the parts needed to be gathered and reports generated from this information. One of the most basic needs was for a simple listing of all the parts, categorized by their TLCSCs. Size (number of lines of code) data was also needed.

The sizing information report was originally produced on an IBM PC. Two line count utilities written by a member of the parts team provided the input to this report. The first line counter simply counted the number of lines of Ada code in a file. This soon proved to be inadequate, however, since more detailed information was needed. There was a need for a separate line count for specifications and bodies, and a separate count of CAMP header comments and comments embedded in the code. Although a single file often contained more than one Ada structure, the original line counter only gave a total for all the Ada structures. An advanced code counter was developed that analyzed a file's Ada structure and kept separate counts for each Ada structure for both the specifications and bodies. Since the first counter did no analyzing of the Ada structure, it ran considerably faster and remained in use for Ada files containing single Ada structures.

Although these tools automated the information gathering, the information itself was still entered into the report by editing the report file. This meant that each time the parts were updated, a new part was added, or the structure of the parts changed, the report file had to be edited again. This was cumbersome because the formatting had to be manually redone every time the file was edited. As a result, information changes were not made as quickly as required and the report became out of date.

In order to address these difficulties, an ORACLE data base was developed to store this information. Reports can now be generated through the use of SQL*Report, an ORACLE utility which allows the generation of reports. SQL*Forms was used as to facilitate data entry.

2. ORACLE RELATIONS

ORACLE is a relational data base with information stored in tables. The parts' sizes were stored in two tables. The first table, named TLCSC, stored information about the TLCSCs. The second table, named AdaLevel, stored information about all of the lower-level Ada structures. The information about the TLCSCs and lower structures varied slightly, which is why separate tables were created.

The TLCSC relation (table), its fields and descriptions are given in the Table A-1 and the AdaLevel relation, its fields and descriptions are given in Table A-2.

TABLE A-1. COLUMNS IN THE TLCSC RELATION

TLCSC Relation	
Column Name	Description
Partno	This is the surrogate part number. Each entry was assigned an arbitrary number to be used as the prime key for that entry.
Tname	TLCSC name of the Ada structure
Require	Requirement number (reference SRS)
Type	Type of Ada structure (procedure, generic package etc.)
Parent	The part number of its parent in the Ada hierarchy. This uses the surrogate numbering scheme as used by the partno field.
Speccodesize	Number of lines of specification code
Bodycodesize	Number of lines of body code
Speccomsize	Number of lines of the header for the spec
Bodycomsize	Number of lines of the body comments
Testcodesize	Number of lines of test code
Part	Indicates whether or not the entry is a part
Used	Indicates whether or not this entry was used by the 11th Missile Application
Subcategory	Subcategory to which TLCSC belongs

TABLE A-2. COLUMNS IN THE ADALEVEL RELATION

AdaLevel Relation	
Column Name	Description
Partno	This is the surrogate part number. Each entry was assigned an arbitrary number to be used as the prime key for that entry.
LIname	Ada name of LLCSC or unit
Require	Requirement number (reference SRS)
Type	Type of Ada structure (procedure, generic package etc.)
Parent	The part number of its parent in the Ada hierarchy. This uses the surrogate numbering scheme as used by the partno field.
Speccodesize	Number of lines of the specification code
Bodycodesize	Number of lines of the body code
Speccomsize	Number of lines of the header for the specification
Bodycomsize	Number of lines of body comments
Part	Indicates whether or not the entry is a part
Used	Indicates whether or not this entry was used by the 11th Missile Application
Levelnum	Hierarchy number for this unit

These tables were used to generate two reports. The first is a list of all the parts divided into their source file components. Along with this list is sizing information, whether it is a part, and whether it was used in the 11th Missile Application. The second report is a list of all the parts used in the 11th Missile Application and their respective code sizes. The second report is discussed in the Appendix of Volume 2.

The parts size report is contained in Table A-3.

TABLE A-3. CAMP PARTS SIZING LIST

(1 of 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Comment Size		Part	11th Use
		Spec	Body	Test	Spec	Body		
P001	Common Navigation Parts	10	10	2,215	215	108	N	Y
	Altitude Integration	12	7		104	137	Y	Y
	Reinitialize	2	7		0	118	N	Y
	Integrate	3	13		0	127	N	Y
	Compute Ground Velocity	10	8		83	107	Y	Y
	Compute Gravitational Acceleration Lat In	22	13		119	172	Y	N
	Compute Gravitational Acceleration Sin Lat In	19	13		114	156	Y	Y
	Compute Heading	10	6		84	113	Y	N
	Update Velocity	20	8		135	183	Y	Y
	Reinitialize	1	5		0	102	N	Y
	Update	4	16		0	172	N	Y
	Current Velocity	1	5		0	83	N	Y
	Scalar Velocity	9	6		84	101	Y	N
	Compute Rotation Increments	11	8		90	112	Y	N
SUBTOTALS		124	115	2,215	813	1,683	8	9
P002	Wander Azimuth Navigation Parts	16	16	677	234	119	N	Y
	Compute Earth Relative Horizontal Velocities	16	16		108	132	Y	N
	Compute Total Angular Velocity	12	7		98	118	Y	N
	Compute Coriolis Acceleration	19	12		121	156	Y	Y
	Total Platform Rotation Rate	11	9		90	119	Y	Y
	Earth Rotation Rate	12	7		121	160	Y	Y
	Compute	4	11		0	11	N	Y
	Compute Earth Relative Navigation Rotation Rate	18	15		116	170	Y	Y
	Compute Wander Azimuth Angle	12	7		101	125	Y	N
	Compute Latitude	7	6		70	96	Y	N
	Compute Latitude Using Arctan	16	12		108	141	Y	N
	Compute East Velocity with Sin Cos In	11	13		97	118	Y	Y
	Compute Longitude	13	7		97	119	Y	N
	Compute Curvatures	31	30		131	216	Y	Y
	Compute East Velocity	14	12		101	130	Y	N
	Compute North Velocity	14	12		101	133	Y	N
	Coriolis Acceleration from Total Rates	12	7		121	160	Y	N
	Compute	4	11		0	6	N	N
	Compute North Velocity with Sin Cos In	11	13		99	123	Y	Y
	Compute Earth Relative Horizontal Velocities With Sin Cos In	13	15		106	119	Y	N
	Compute Latitude Using Two Value Arctangent	14	16		100	119	Y	Y
	Compute Longitude using Two Value Arctangent	11	11		88	105	Y	Y
	Compute Wander Azimuth Angle using Two Value Arctangent	10	12		92	115	Y	Y
SUBTOTALS		285	261	677	2,066	2,691	20	11
P003	North Pointing Navigation Parts	9	9	541	190	117	N	N
	Compute Coriolis Acceleration	17	14		102	142	Y	N
	Total Platform Rotation Rates	11	9		82	104	Y	N
	Earth Rotation Rate	16	7		114	148	Y	N
	Compute	2	12		0	6	N	N
	Earth Relative Navigation Rotation Rate	18	7		134	155	Y	N
	Compute	4	11		0	6	N	N
	Latitude Integration	13	6		100	147	Y	N
	Reinitialize	2	7		0	121	N	N
	Integrate	3	12		0	131	N	N
	Longitude Integration	18	43		112	164	Y	N
	Reinitialize	3	8		0	134	N	N
	Integrate	4	16		0	149	N	N
	Radius of Curvature	25	7		127	193	Y	N
	Compute	2	25		0	6	N	N
SUBTOTALS		138	184	541	771	1,606	7	0
P361	General Utilities	3	3	0	69	63	N	N
	Instruction Set Test	6	6		75	90	Y	N
SUBTOTALS		6	6	0	75	90	1	0
P601	Asynchronous Control	2	0	0	0	0	Y	N
	Data Driven Task Shell	2	0		0	0	N	N

TABLE A-3. CAMP PARTS SIZING LIST (2 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Test	Comment Size			Part	11th Use
		Spec	Body			Spec	Body			
	Interrupt-Driven Task Shell	5	0	0		0	0		N	N
	Aperiodic Task Shell	3	0	0		0	0		N	N
	Continuous Task Shell	2	0	0		0	0		N	N
	Periodic Task Shell	3	0	0		0	0		N	N
SUBTOTALS		15	0	0		0	0		0	0
P602	Communication Parts	3	3	296		81	73		N	N
	Update Exclusion	9	3			126	95		Y	N
	Read Update	5	29			0	0		N	N
	Attempt Read	2	10			0	0		N	N
	Attempt Read Wait	2	6			0	0		N	N
	Attempt Read Delay	3	12			0	0		N	N
	Attempt Start Update	3	13			0	0		N	N
	Attempt Start Update Wait	3	8			0	0		N	N
	Attempt Start Update Delay	4	14			0	0		N	N
	Attempt Complete Update	3	15			0	0		N	N
SUBTOTALS		34	110	296		125	95		1	0
P611	WGS72 Ellipsoid Metric Data	29	0	96		125	0		Y	M
P612	WGS72 Ellipsoid Engineering Data	30	0	92		143	0		Y	M
P613	WGS72 Ellipsoid Unitless Data	11	0	160		70			Y	Y
P614	Conversion Factors	41	0	200		121	0		Y	Y
P615	Universal Constants	9	0	129		72	0		Y	Y
P621	Basic Data Types	135	185	331		182	436		Y	M
P622	Kalman Filter Data Types	213	40	186		387	127		Y	N
P623	Autopilot Data Types	88	92	267		145	228		Y	N
P631	Missile Radar Altimeter Handler Parts	15	25	0		220	3		Y	N
	Power On	1	22			0	17		N	N
	Power Off	1	4			0	0		N	N
	Goto Transmit Mode	1	4			0	0		N	N
	Goto Standby Mode	1	4			0	0		N	N
	Perform Built In Test	4	7			0	0		N	N
	Perform Built In Test Sequence	4	7			0	0		N	N
	Read Altitude Feet	4	7			0	0		N	N
	Read Altitude Integer	4	7			0	0		N	N
SUBTOTALS		20	62	0		0	17		0	0
P632	Missile Radar Altimeter Handler Auto	15	25	0		190	1		Y	N
	Goto Transmit Mode	1	22			0	17		N	N
	Goto Standby Mode	1	4			0	0		N	N
	Perform Built In Test	4	7			0	0		N	N
	Perform Built In Test Sequence	4	7			0	0		N	N
	Read Altitude Feet	4	7			0	0		N	N
	Read Altitude Integer	4	7			0	0		N	N
SUBTOTALS		18	54	0		0	17			0
P633	Bus Interface Parts	6	0	0		229	0		Y	M
	Send Message Using Address No Wait	4	0			0	0		N	N
	Send Message Using Address Wait	5	0			0	0		N	N
	Data Transfer No Wait	5	0			0	0		N	N
	Data Transfer Wait	6	0			0	0		N	N
	Perform Built In Test	3	0			0	0		N	N
	Interface	18	0			6	0		N	N
	Update Retry Count	1	0			0	0		N	N
	Send Command Wait	3	0			0	0		N	N

TABLE A-3. CAMP PARTS SIZING LIST (3 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Comment Size			Part	11th Use
		Spec	Body	Test	Spec	Body			
	Send Message No Wait	4	0		0	0		N	M
	Send Message Wait	5	0		0	0		N	N
	SUBTOTALS	54	0	0	6	0		0	0
P634	Clock Handler	5	11	203	132	121		Y	Y
	Current Time	1	5		0	98		N	Y
	Converted Time	2	6		0	103		N	N
	Reset Clock	1	5		0	94		N	N
	Synchronize Clock	3	7		0	99		N	Y
	Elapsed Time	1	9		0	112		N	Y
	SUBTOTALS	8	32	203	0	506		0	2
P644	Direction Cosine Matrix Operations	5	5	541	158	103		N	Y
	DCM General Operations	11	11		86	92		N	Y
	DCM Initialized From Reference	23	62		203	221		Y	Y
	DCM Trapezoidal Integration	26	7		205	201		Y	N
	Reinitialize Angular Velocities	3	8		0	129		N	N
	Perform Trapezoidal Integration of DCM	5	44		0	253		N	N
	Perform Rectangular Integration of DCM	24	27		174	199		Y	Y
	Reorthonormalize DCM	23	33		169	184		Y	Y
	Frame Misalignment	29	18		188	194		Y	Y
	Aligned DCM Matrix	29	26		186	213		Y	Y
	DCM From Quaternion	26	37		167	193		Y	Y
	Compute First Row from Orthonormal	16	11		141	135		Y	Y
	CNE Operations	29	26		227	213		N	Y
	Reorthonormalize CNE	1	4		0	0		Y	Y
	CNE Initialized From Earth Position	15	28		116	157		Y	Y
	CNE Integration	14	21		135	233		Y	Y
	Perform Trapezoidal Integration of CNE	5	9		0	0		N	Y
	Reinit Ang Vel For Trapez Integ of CNE	3	8		0	0		N	Y
	Perform Rectangular Integration of CNE	5	7		0	0		N	Y
	Alignment Parts	18	21		160	205		Y	Y
	Frame Misalignment of CNE	4	7		0	0		N	Y
	Aligned CNE Matrix	4	7		0	0		N	Y
	CNE From Quaternion	13	11		140	160		Y	Y
	Compute CNE	3	6		0	0		N	Y
	Compute First Row of CNE From Orthonormal	2	5		0	0		Y	Y
	CNE Initialized From Reference	8	17		0	0		Y	Y
	SUBTOTALS	339	461	541	2,297	2,982		15	22
P651	Kalman Filter Common Parts	6	6	406	116	65		N	Y
	State Transition And Process Noise Matrices								
	Manager	31	10		142	140		Y	Y
	Initialize	1	6		0	91		N	Y
	Propagate	3	10		0	130		N	Y
	Get_Current	3	7		0	101		N	Y
	Propagated_Phi	1	5		0	90		N	Y
	Error Covariance Matrix Manager	15	8		97	110		Y	Y
	Initialize	1	5		0	80		N	Y
	Propagate	2	6		0	98		N	Y
	P	1	5		0	79		N	Y
	State Transition Matrix Manager	11	8		95	104		Y	N
	Propagated_Phi	1	5		0	90		N	N
	Initialize	1	5		0	73		N	N
	Propagate	1	5		0	93		N	N
	SUBTOTALS	72	85	406	334	1,279		3	9
P652	Kalman Filter Compact H Parts	8	25	462	136	69		N	Y
	Compute Kalman Gain	19	18		96	100		Y	Y
	Update Error Covariance Matrix	22	12		96	99		Y	N
	Update State Vector	20	13		95	94		Y	Y
	Sequentially Update Covariance Matrix and State								
	Vector	30	28		117	134		Y	N
	Update	6	24		0	104		N	N

TABLE A-3. CAMP PARTS SIZING LIST (4 OF 14)

TLCS No.	TLCS Name Lower Level Units	Code Size			Comment Size			Part	11th Uac
		Spec	Body	Test	Spec	Body			
	Kalman Update	44	21		147	162	Y	N	
	Update	8	23		0	117	N	N	
	Update Error Covariance Matrix General Form	29	17		123	128	Y	Y	
SUBTOTALS		178	156	462	674	938	6	3	
P653	Kalman Filter Complicated H Parts	8	21	457	130	72	N	Y	
	Compute Kalman Gain	30	20		113	123	Y	Y	
	Update Error Covariance Matrix	25	13		110	113	Y	N	
	Update State Vector	26	14		105	110	Y	Y	
	Sequentially Update Covariance Matrix and State Vector								
	Update	40	32		134	156	Y	N	
	Kalman Update	6	24		0	106	N	N	
	Update	53	20		161	182	Y	N	
	Update Error Covariance Matrix General Form	7	22		0	112	N	N	
	Update Error Covariance Matrix General Form	35	17		126	126	Y	Y	
SUBTOTALS		222	162	457	749	1,028	6	3	
P661	Waypoint Steering	13	28	1,022	176	105	N	Y	
	Distance to Current Waypoint	15	11		111	116	Y	N	
	Compute Turning and Nonturning Distances	12	14		96	129	Y	Y	
	Turn Test Operations	5	14		85	93	Y	Y	
	Stop Test	4	13		0	110	N	Y	
	Start Test	4	13		0	104	N	Y	
	Steering Vector Operation	22	40		170	176	Y	N	
	Initialize	12	41		0	174	N	N	
	Update	8	24		0	150	N	N	
	Steering Vector Operations with Arcsin	24	23		174	167	Y	Y	
	Initialize	12	40		0	171	N	Y	
	Update	8	23		0	147	N	Y	
	Compute Turn Angle and Direction	18	24		116	155	Y	Y	
	Crosstrack and Heading Error Operations	37	33		187	169	Y	Y	
	Compute When not Turning	6	23		0	185	N	N	
	Compute	12	36		0	158	N	N	
	Compute When Turning	11	43		0	228	N	Y	
	Distance to Current Waypoint with Arcsin	19	11		117	150	Y	Y	
SUBTOTALS		229	426	1,022	1,056	2,582	8	11	
P662	Autopilot	5	6	2,553	146	64	N	Y	
	Integral Plus Proportional Gain	12	7		115	100	Y	N	
	Integrate	1	5		0	141	N	N	
	Update Proportional Gain	1	5		0	112	N	N	
	Pitch Autopilot	47	31		233	143	Y	N	
	Initialize Pitch Autopilot	5	21		0	138	N	N	
	Compute Elevator Command	0	23		0	151	N	N	
	Update Pitch Rate Gain	0	5		0	89	N	N	
	Update Acceleration Gain	0	5		0	91	N	N	
	Update Integrator Gain	0	5		0	98	N	N	
	Update Integrator Limit	0	5		0	98	N	N	
	Update Proportional Gain	2	7		0	99	N	N	
	Lateral Directional Autopilot	64	59		392	251	Y	N	
	Initialize Lateral Directional Autopilot	10	38		0	207	N	N	
	Compute Aileron Rudder Commands	9	42		0	230	N	N	
	Update Aileron Integrator Gain	2	6		0	96	N	N	
	Update Aileron Integrator Limit	2	7		0	99	N	N	
	Update Roll Command Proportional Gain	2	6		0	97	N	N	
	Update Roll Rate Gain For Aileron	2	6		0	91	N	N	
	Update Yaw Rate Gain For Aileron	2	6		0	91	N	N	
	Update Rudder Integrator Gain	2	6		0	98	N	N	
	Update Rudder Integrator Limit	2	7		0	98	N	N	
	Update Feedback Rate Gain For Rudder	2	6		0	91	N	N	
	Update Roll Rate Gain For Rudder	2	6		0	91	N	N	
	Update Acceleration Proportional Gain	2	6		0	98	N	N	
SUBTOTALS		171	320	2,553	740	2,898	3	0	

TABLE A-3. CAMP PARTS SIZING LIST (5 OF 14)

TL.CSC No.	TL.CSC Name Lower Level Units	Code Size				Comment Size			Part	11th	
		Spec	Body	Test		Spec	Body				
P671	Air Data Parts	9	23	28R		90	61		N	N	
	Compute Outside Air Temperature	16	9			100	99		Y	N	
	Compute Pressure Ratio	12	11			92	90		Y	N	
	Compute Mach	12	6			95	94		Y	N	
	Compute Dynamic Pressure	11	9			85	85		Y	N	
	Compute Speed of Sound	13	7			91	93		Y	N	
	Barometric Altitude Integration	19	8			115	108		Y	N	
	Compute Barometric Altitude	4	27			0	121		N	N	
	SUBTOTALS	87	77	28R		578	690		6	0	
P672	Fuel Control Parts	4	6	402		84	71		N	N	
	Throttle Command Manager	20	62			117	211		Y	N	
	Compute Throttle Command	4	17			0	117		N	N	
	Update Mach Error Limit	2	5			0	81		N	N	
	Update Mach Error Integral Limit	2	6			0	82		N	N	
	Update Throttle Rate Limit	2	6			0	82		N	N	
	Update Throttle Command Limits	3	8			0	84		N	N	
	Update Mach Error Gain	2	5			0	80		N	N	
	Update Throttle Bandwidth	2	6			0	82		N	N	
	SUBTOTALS	37	115	402		117	819		1	0	
P681	Coordinate Vector Matrix Algebra	10	17	858		109	94		N	Y	
	Matrix Operations	8	20			101	90		N	N	
	"+"	2	16			0	70		Y	N	
	"."	2	16			0	70		Y	N	
	"x"	2	16			0	72		Y	N	
	"-"	2	16			0	72		Y	N	
	Set_to_Identity_Matrix	1	6			0	50		Y	N	
	Set_to_Zero_Matrix	1	6			0	51		Y	N	
	Vector Scalar Operations	14	9			135	107		N	Y	
	"o"	2	10			0	105		Y	Y	
	Sparse_X_Vector_Scalar_Multiply	3	11			0	121		Y	N	
	"f"	2	10			0	110		Y	Y	
	Matrix Scalar Operations	14	9			116	69		N	N	
	"o"	2	16			0	72		Y	N	
	"f"	2	16			0	71		Y	N	
	Cross Product	14	14			128	75		Y	Y	
	Matrix Vector Multiply	16	22			124	74		Y	N	
	Matrix Matrix Multiply	14	39			110	79		Y	N	
	Vector Operations	11	21			130	165		N	Y	
	Sparse_Right_XY_Subtract	2	10			0	111		Y	Y	
	Set_to_Zero_Vector	1	5			0	98		Y	N	
	"+"	2	10			0	115		Y	Y	
	"."	2	10			0	114		Y	Y	
	Vector_Length	1	5			0	111		Y	N	
	Dot_Product	2	10			0	112		Y	Y	
	Sparse_Right_Z_Add	2	10			0	112		Y	Y	
	Sparse_Right_X_Add	2	10			0	111		Y	N	
	SUBTOTALS	126	343	858		844	2,407		22	10	
P682	General Vector Matrix Algebra	33	43	3,792		447	177		N	Y	
	ABA_Trans_Dynam_Sparse_Matrix_Sq_Matrix	16	50			133	148		N	N	
	ABA_Transpose	3	12			0	6		Y	N	
	ABA_Trans_Vector_Sq_Matrix	16	33			121	139		N	N	
	ABA_Transpose	2	11			0	6		Y	N	
	ABA_Trans_Vector_Scalar	14	41			120	164		N	N	
	ABA_Transpose	2	10			0	6		Y	N	
	Column_Matrix_Operations	12	7			123	100		N	Y	
	Set_Diagonal_and_Subtract_from_Identity	3	17			0	93		Y	Y	
	ABA_Transpose	9	33			0	121		Y	N	
	ABA_Symm_Transpose	9	37			0	133		Y	Y	
	Dot Product Operations Unrestricted	13	9			131	120		N	N	
	Dot Product	2	19			0	102		Y	N	
	Dot Product Operations Restricted	13	14			118	115		Y	N	
	Diagonal_Full_Matrix_Add_Unrestricted	15	12			149	118		N	N	
	"+"	2	38			0	135		Y	N	
	Diagonal_Full_Matrix_Add_Restricted	10	21			112	103		Y	Y	

TABLE A-3. CAMP PARTS SIZING LIST (6 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Spec	Code Size Body	Test	Comment Size Spec Body	Part	11th Use
	Matrix Scalar Operations Constrained	15	5		135	97	N
	"e"	2	14		0	107	Y
	"f"	2	14		0	109	Y
	Diagonal Matrix Scalar Operations	15	11		182	113	N
	"e"	2	18		0	104	Y
	"f"	2	18		0	97	Y
	Matrix Vector Multiply Unrestricted	21	10		275	149	N
	"e"	2	31		0	133	Y
	Matrix Vector Multiply Restricted	19	18		252	121	Y
	Vector Matrix Multiply Unrestricted	21	10		273	160	N
	"e"	2	27		0	6	Y
	Vector Matrix Multiply Restricted	19	16		254	130	Y
	Vector Vector Transpose Multiply Unrestricted	20	10		155	136	N
	"e"	2	28		0	129	Y
	Vector Vector Transpose Multiply Restricted	16	16		136	113	Y
	Matrix Matrix Multiply Unrestricted	25	11		266	147	N
	"e"	2	41		0	141	Y
	Matrix Matrix Multiply Restricted	18	20		240	123	Y
	Matrix Matrix Transpose Multiply Unrestricted	23	11		150	113	N
	"e"	2	41		0	140	Y
	Matrix Matrix Transpose Multiply Restricted	16	21		131	117	Y
	Symmetric Full Storage Matrix Operations						
	Constrained	8	15		124	102	N
	Change Element	4	17		0	111	Y
	Set_to_Identity_Matrix	1	16		0	101	Y
	Set_to_Zero_Matrix	1	5		0	83	Y
	Add_to_Identity	1	18		0	100	Y
	Subtract_from_Identity	1	28		0	131	Y
	"+"	2	30		0	126	Y
	"-"	2	30		0	126	Y
	Diagonal Matrix Operations	13	47		153	200	N
	Identity_Matrix	1	5		0	95	Y
	Zero_Matrix	1	5		0	94	Y
	Change_Element	4	12		0	133	Y
	Retrieve_Element	3	10		0	121	Y
	Row_Slice	2	12		0	137	Y
	Column_Slice	2	12		0	139	Y
	Add_to_Identity	2	11		0	113	Y
	Subtract_from_Identity	2	11		0	113	Y
	"+"	2	11		0	112	Y
	"-"	2	11		0	114	Y
	Vector Scalar Operations Unconstrained	15	5		142	117	N
	"e"	2	20		0	129	Y
	"f"	2	20		0	129	Y
	Vector Scalar Operations Constrained	14	5		139	95	N
	"e"	2	11		0	103	Y
	"f"	2	11		0	103	Y
	Matrix Scalar Operations Unconstrained	19	5		141	118	N
	"e"	2	34		0	134	Y
	"f"	2	34		0	135	Y
	Symmetric Half Storage Matrix Operations	12	53		148	211	N
	Initialize	3	19		0	145	N
	Identity_Matrix	1	5		0	87	Y
	Zero_Matrix	1	5		0	85	Y
	Change_Element	4	14		0	132	Y
	Retrieve_Element	3	15		0	135	Y
	Row_Slice	2	18		0	134	Y
	Column_Slice	2	18		0	137	Y
	Add_to_Identity	1	13		0	138	Y
	Subtract_from_Identity	1	16		0	143	Y
	"+"	2	11		0	143	Y
	"-"	2	11		0	144	Y
	Swap_Col	0	7		0	83	N
	Swap_Row	0	7		0	83	N
	Symmetric Full Storage Matrix Operations						
	Unconstrained	10	10		124	99	N
	Change_Element	4	24		0	155	Y
	Set_to_Identity_Matrix	1	20		0	141	Y
	Set_to_Zero_Matrix	1	5		0	89	Y
	Add_to_Identity	1	22		0	133	Y

TABLE A-3. CAMP PARTS SIZING LIST (7 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Test	Comment Size			Part	11th
		Spec	Body			Spec	Body			Use
	Subtract_from_Identity	1	40			0	174		Y	N
	"+"	2	48			0	152		Y	N
	"."	2	48			0	151		Y	N
	Matrix Operations Unconstrained	9	11			131	113		N	N
	"+"	2	34			0	138		Y	N
	"."	2	34			0	137		Y	N
	"+"	2	14			0	112		Y	N
	"."	2	14			0	112		Y	N
	Set_to_Identity_Matrix	1	20			0	142		Y	N
	Set_to_Zero_Matrix	1	5			0	97		Y	N
	"*"	2	38			0	150		Y	N
	Matrix Operations Constrained	9	9			115	96		N	N
	"+"	2	15			0	97		Y	N
	"."	2	15			0	98		Y	N
	"+"	2	14			0	97		Y	N
	"."	2	14			0	97		Y	N
	Set_to_Identity_Matrix	1	20			0	119		Y	N
	Set_to_Zero_Matrix	1	5			0	83		Y	N
	Dynamically Sparse Matrix Operations									
	Unconstrained	9	9			110	100		N	N
	Set_to_Identity_Matrix	1	20			0	137		Y	N
	Set_to_Zero_Matrix	1	5			0	96		Y	N
	Add_to_Identity	1	26			0	132		Y	N
	Subtract_from_Identity	1	33			0	132		Y	N
	"+"	2	44			0	138		Y	N
	"."	2	44			0	137		Y	N
	Dynamically Sparse Matrix Operations Constrained	8	9			110	85		N	N
	Set_to_Zero_Matrix	1	5			0	84		Y	N
	Add_to_Identity	1	26			0	117		Y	N
	Subtract_from_Identity	1	33			0	118		Y	N
	"+"	2	25			0	105		Y	N
	"."	2	25			0	108		Y	N
	Set_to_Identity_Matrix	1	20			0	117		Y	N
	Vector Operations Unconstrained	13	8			131	115		N	N
	"+"	2	22			0	125		Y	N
	"."	2	22			0	124		Y	N
	Dot_Product	2	12			0	138		Y	N
	Vector_Length	1	23			0	149		Y	N
	Vector Operations Constrained	13	7			120	105		N	Y
	Dot_Product	2	12			0	131		Y	N
	Vector_Length	1	12			0	113		Y	N
	"+"	2	11			0	97		Y	Y
	"."	2	11			0	98		Y	N
SUBTOTALS		670	2,361	3,792		5,144	14,791		97	17
P6R3	Standard Trig	13	79	685		189	240		N	Y
	Arctan2	11	27			0	125		Y	Y
	Sin	1	5			0	93		Y	N
	Sin	1	4			0	0		Y	N
	Sin	1	4			0	0		Y	N
	Cos	1	5			0	93		Y	N
	Cos	1	4			0	0		Y	N
	Cos	1	4			0	0		Y	N
	Sin_Cos	3	32			0	140		Y	N
	Sin_Cos	3	26			0	11		Y	N
	Sin_Cos	3	26			0	11		Y	N
	Tan	1	5			0	93		Y	N
	Tan	1	4			0	0		Y	N
	Tan	1	4			0	0		Y	N
	Arcsin	1	5			0	91		Y	N
	Arcsin	1	4			0	0		Y	N
	Arcsin	1	4			0	0		Y	N
	Arccos	1	5			0	91		Y	N
	Arccos	1	4			0	0		Y	N
	Arccos	1	4			0	0		Y	N
	Arcsin_Arccos	3	10			0	117		Y	N
	Arcsin_Arccos	3	9			0	0		Y	N
	Arcsin_Arccos	3	9			0	0		Y	N
	Arctan	1	5			0	91		Y	N

TABLE A-3. CAMP PARTS SIZING LIST (8 OF 14)

TLCS No.	TLCS Name Lower Level Units	Code Size			Comment	Size		Part	11th	Use
		Spec	Body	Test		Spec	Body			
	Arctan	1	4		0	0	Y	N		
	Arctan	1	4		0	0	Y	N		
SUBTOTALS		47	217	685	0	956	25	1		
P684	Geometric Operations	7	21	392	109	93	N	Y		
	Unit Radial Vector	15	22		107	134	Y	Y		
	Unit Normal Vector	14	13		101	123	Y	N		
	Compute Segment and Unit Normal Vector	21	16		126	140	Y	N		
	Compute Segment and Unit Normal Vector w/ Arcsin	23	16		130	135	Y	Y		
	Great Circle Arc Length	14	29		122	212	Y	N		
	Compute	4	18		0	8	N	N		
SUBTOTALS		91	114	392	586	752	5	2		
P686	Signal Processing	14	18	1,180	225	102	N	Y		
	General First Order Filter	18	14		108	149	Y	M		
	Update Coefficients	0	9		0	99	N	N		
	Filter	0	11		0	115	N	N		
	Reinitialize	0	8		0	81	N	N		
	Tustin Lead Lag Filter	14	12		105	148	Y	N		
	Update Coefficients	0	7		0	95	N	N		
	Filter	0	12		0	120	N	N		
	Reinitialize	0	6		0	77	N	N		
	Tustin Lag Filter	14	13		104	143	Y	N		
	Update Coefficients	0	7		0	95	N	N		
	Filter	0	10		0	126	N	N		
	Reinitialize	0	9		0	81	N	N		
	Second Order Filter	16	25		113	156	Y	N		
	Redefine Coefficients	0	17		0	101	N	N		
	Filter	0	15		0	123	N	N		
	Reinitialize	0	8		0	84	N	N		
	Tustin Integrator With Limit	16	26		132	230	Y	N		
	Update Limit	1	5		0	109	N	N		
	Update Gain	1	5		0	100	N	N		
	Integrate	1	29		0	175	N	N		
	Reset	2	10		0	97	N	N		
	Limit Flag Setting	1	5		0	72	N	N		
	Tustin Integrator With Asymmetric Limit	17	29		139	160	Y	N		
	Update Limits	2	9		0	74	N	N		
	Update Gain	1	5		0	67	N	N		
	Integrate	1	36		0	138	N	N		
	Reset	2	10		0	81	N	N		
	Limit Flag Setting	1	5		0	62	N	N		
	Upper Lower Limiter	8	12		91	129	Y	Y		
	Update Limits	2	11		0	97	N	Y		
	Limit	1	13		0	92	N	Y		
	Upper Limiter	6	7		79	93	Y	Y		
	Update Limit	1	5		0	85	N	Y		
	Limit	1	11		0	94	N	Y		
	Lower Limiter	6	7		79	98	Y	Y		
	Update Limit	1	6		0	85	N	Y		
	Limit	1	11		0	94	N	Y		
	Absolute Limiter	6	7		80	107	Y	Y		
	Update Limit	1	5		0	92	N	Y		
	Limit	1	15		0	99	N	Y		
	Absolute Limiter With Flag	6	9		86	121	Y	N		
	Limit Flag Setting	1	5		0	76	N	N		
	Limit	1	18		0	106	N	N		
	Update Limit	1	5		0	95	N	N		
SUBTOTALS		152	504	1,180	1,116	4,721	11	12		
P687	General Purpose Math	18	31	1,061	178	82	N	Y		
	Integrator	12	9		91	115	Y	N		
	Reinitialize	2	7		0	65	N	N		
	Update	1	5		0	60	N	N		
	Integrate	4	11		0	75	N	N		
	Interpolate or Extrapolate	14	14		82	133	Y	N		
	Square Root	7	8		79	110	Y	M		

TABLE A-3. CAMP PARTS SIZING LIST (9 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Comment Size			Part	11th Use
		Spec	Body	Test	Spec	Body			
	Sqrt	1	6		0	0	N	M	
	Root Sum Of Squares	11	9		84	89	Y	Y	
	Sign	5	12		68	92	Y	N	
	Mean Value	6	10		75	99	Y	N	
	Mean Absolute Difference	7	19		75	104	Y	N	
	Two Way Table Lookup	28	12		127	111	Y	N	
	Initialize	4	8		0	0	N	N	
	Lookup Y	2	54		0	0	N	N	
	Lookup X	2	54		0	0	N	N	
	Lookup Table Even Spacing	12	14		130	148	Y	N	
	Initialize	3	7		0	59	N	N	
	Lookup	6	33		0	107	N	N	
	Lookup	7	46		0	108	N	N	
	Lookup Table Uneven Spacing	15	7		120	107	Y	N	
	Initialize	4	10		0	60	N	N	
	Lookup	6	24		0	94	N	N	
	Lookup	7	36		0	97	N	N	
	Incrementor	7	8		79	102	Y	N	
	Reinitialize	2	7		0	71	N	N	
	Increment	1	6		0	57	N	N	
	Decrementor	7	8		79	105	Y	N	
	Reinitialize	2	7		0	62	N	N	
	Decrement	1	6		0	57	N	N	
	Running Average	9	9		83	108	Y	N	
	Reinitialize	2	7		0	61	N	N	
	Reinitialize	1	5		0	56	N	N	
	Current Average	1	7		0	59	N	N	
	Accumulator	6	9		77	80	Y	Y	
	Reinitialize	1	5		0	57	N	Y	
	Accumulate	1	5		0	56	N	Y	
	Accumulate	2	8		0	63	N	Y	
	Retrieve	1	5		0	60	N	Y	
	Change Calculator	6	8		73	87	Y	N	
	Reinitialize	1	5		0	57	N	N	
	Change	1	8		0	73	N	N	
	Retrieve Value	1	5		0	54	N	N	
	Change Accumulator	7	12		86	109	Y	N	
	Reinitialize	1	5		0	57	N	N	
	Reinitialize	2	7		0	52	N	N	
	Accumulate Change	1	6		0	65	N	N	
	Accumulate Change	2	9		0	70	N	N	
	Retrieve Accumulation	1	5		0	54	N	N	
	Retrieve Previous Value	1	5		0	54	N	N	
SUBTOTALS		234	592	1,061	1,408	3,629	16	6	
P688	Polynomials	13	15	4,813	254	129	N	Y	
	Reduction Operations	7	4		95	99	N	Y	
	Sine Reduction	1	13		0	74	Y	Y	
	Cosine Reduction	1	6		0	74	Y	Y	
	Taylor Series	6	8		129	82	N	N	
	Taylor Natural Log	8	17		89	108	N	N	
	Nat Log 8term	1	18		0	0	Y	N	
	Nat Log 7term	1	17		0	0	Y	N	
	Nat Log 6term	1	16		0	0	Y	N	
	Nat Log 5term	1	15		0	0	Y	N	
	Nat Log 4term	1	14		0	0	Y	N	
	Taylor Log Base N	9	6		99	110	N	N	
	Log Base N 8term	2	4		0	0	Y	N	
	Log N 8term	1	4		0	0	N	N	
	Log Base N 7term	2	4		0	0	Y	N	
	Log N 7term	1	4		0	0	N	N	
	Log Base N 6term	2	4		0	0	Y	N	
	Log N 6term	1	4		0	0	N	N	
	Log Base N 5term	2	4		0	0	Y	N	
	Log N 5term	1	4		0	0	N	N	
	Log Base N 4term	2	4		0	0	Y	N	
	Log N 4term	1	4		0	0	N	N	
	Taylor Degree Operations	10	54		151	206	N	N	
	Mod Cos D 4term	1	32		0	0	Y	N	

TABLE A-3. CAMP PARTS SIZING LIST (10 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Test	Comment Size			Part	11th Use
		Spec	Body			Spec	Body			
	Tan D 8term	1	17			0	0		Y	N
	Mod Tan D 8term	1	5			0	0		Y	N
	Mod Tan D 7term	1	5			0	0		Y	N
	Mod Tan D 6term	1	5			0	0		Y	N
	Mod Tan D 5term	1	5			0	0		Y	N
	Mod Tan D 4term	1	5			0	0		Y	N
	Sin D 8term	1	17			0	0		Y	N
	Sin D 7term	1	16			0	0		Y	N
	Sin D 6term	1	15			0	0		Y	N
	Sin D 5term	1	14			0	0		Y	N
	Mod Sin D 8term	1	34			0	0		Y	N
	Mod Sin D 7term	1	32			0	0		Y	N
	Mod Sin D 6term	1	30			0	0		Y	N
	Mod Sin D 5term	1	28			0	0		Y	N
	Mod Sin D 4term	1	26			0	0		Y	N
	Cos D 8term	1	25			0	0		Y	N
	Cos D 7term	1	24			0	0		Y	N
	Cos D 6term	1	23			0	0		Y	N
	Cos D 5term	1	22			0	0		Y	N
	Cos D 4term	0	27			0	0		N	N
	Mod Cos D 8term	1	40			0	0		Y	N
	Mod Cos D 7term	1	38			0	0		Y	N
	Mod Cos D 6term	1	36			0	0		Y	N
	Mod Cos D 5term	1	34			0	0		Y	N
	Sin D 4term	0	19			0	0		N	N
	Taylor Radian Operations	13	90			210	249		N	N
	Arctan R 7term	1	22			0	0		Y	N
	Arctan R 6term	1	21			0	0		Y	N
	Arctan R 5term	1	20			0	0		Y	N
	Arctan R 4term	1	19			0	0		Y	N
	Alt Arctan R 8term	1	16			0	0		Y	N
	Alt Arctan R 7term	1	15			0	0		Y	N
	Alt Arctan R 6term	1	14			0	0		Y	N
	Alt Arctan R 5term	1	13			0	0		Y	N
	Alt Arctan R 4term	1	12			0	0		Y	N
	Mod Sin R 6term	1	29			0	0		Y	N
	Mod Sin R 5term	1	27			0	0		Y	N
	Mod Sin R 4term	1	25			0	0		Y	N
	Cos R 8term	1	25			0	0		Y	N
	Cos R 7term	1	24			0	0		Y	N
	Cos R 6term	1	23			0	0		Y	N
	Cos R 5term	1	22			0	0		Y	N
	Cos R 4term	1	21			0	0		Y	N
	Mod Cos R 8term	1	39			0	0		Y	N
	Mod Cos R 7term	1	37			0	0		Y	N
	Mod Cos R 6term	1	35			0	0		Y	N
	Mod Cos R 5term	1	33			0	0		Y	N
	Mod Cos R 4term	1	31			0	0		Y	N
	Tan R 8term	1	16			0	0		Y	N
	Mod Tan R 8term	1	5			0	0		Y	N
	Mod Tan R 7term	1	5			0	0		Y	N
	Mod Tan R 6term	1	5			0	0		Y	N
	Mod Tan R 5term	1	5			0	0		Y	N
	Mod Tan R 4term	1	5			0	0		Y	N
	Arcsin R 8term	1	16			0	0		Y	N
	Arcsin R 7term	1	15			0	0		Y	N
	Arcsin R 6term	1	14			0	0		Y	N
	Arcsin R 5term	1	13			0	0		Y	N
	Arccos R 8term	1	16			0	0		Y	N
	Arccos R 7term	1	15			0	0		Y	N
	Arccos R 6term	1	14			0	0		Y	N
	Arccos R 5term	1	13			0	0		Y	N
	Arctan R 8term	1	23			0	0		Y	N
	Sin R 8term	1	16			0	0		Y	N
	Sin R 7term	1	15			0	0		Y	N
	Sin R 6term	1	14			0	0		Y	N
	Sin R 5term	1	13			0	0		Y	N
	Sin R 4term	1	12			0	0		Y	N
	Mod Sin R 8term	1	33			0	0		Y	N
	Mod Sin R 7term	1	31			0	0		Y	N

TABLE A-3. CAMP PARTS SIZING LIST (11 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Comment Size		Part	11th Use
		Spec	Body	Test	Spec	Body		
	Modified Newton Raphson	3	8		76	130	N	Y
	SqRt	4	33		0	0	Y	Y
	Newton Raphson	3	9		76	139	N	N
	SqRt	4	33		0	0	Y	N
	System Functions	15	10		72	62	N	N
	Semicircle Operations	13	22		114	144	N	N
	Sin	1	7		0	114	Y	N
	Cos	1	7		0	114	Y	N
	Tan	1	8		0	116	Y	N
	Arcsin	1	8		0	129	Y	N
	Arccos	1	8		0	129	Y	N
	Arctan	1	7		0	123	Y	N
	Degree Operations	7	24		90	122	N	N
	Sin	1	8		0	108	Y	N
	Cos	1	8		0	107	Y	N
	Tan	1	8		0	108	Y	N
	Arcsin	1	8		0	110	Y	N
	Arccos	1	8		0	110	Y	N
	Arctan	1	7		0	105	Y	N
	Square Root	6	8		74	103	Y	N
	Sqrt	1	8		0	108	N	N
	Base 10 Logarithm	7	8		73	103	Y	N
	Log 10	1	8		0	108	N	N
	Base N Logarithm	11	14		97	156	Y	N
	Log N	1	10		0	125	N	N
	Radian Operations	7	21		90	122	N	N
	Sin	1	7		0	104	Y	N
	Cos	1	7		0	104	Y	N
	Tan	1	8		0	108	Y	N
	Arcsin	1	8		0	110	Y	N
	Arccos	1	8		0	110	Y	N
	Arctan	1	7		0	104	Y	N
	Cody Waite	4	6		50	82	N	N
	Cody Natural Log	8	14		82	114	N	N
	Nat Log	1	31		0	0	Y	N
	R	0	6		0	0	N	N
	Defloat	0	42		0	0	N	N
	Cody Log Base N	9	6		91	109	N	N
	Log Base N	2	4		0	0	Y	N
	Log N	1	4		0	0	N	N
	Continued Fractions	3	5		50	76	N	N
	Continued Radian Operations	9	4		98	103	N	N
	Tan R	3	20		0	0	Y	N
	Arctan R	3	25		0	0	Y	N
	Fike	3	5		50	80	N	Y
	Fike Semicircle Operations	8	10		91	128	N	Y
	Arcsin S 6term	1	31		0	0	Y	Y
	Arccos S 6term	1	32		0	0	Y	Y
	General Polynomial	16	4		126	134	N	N
	Polynomial	1	12		0	0	Y	N
	Hart	4	6		55	82	N	N
	Hart Radian Operations	11	9		97	122	N	N
	Cos R 5term	1	22		0	0	Y	N
	Hart Degree Operations	9	9		95	133	N	N
	Cos D 5term	1	22		0	0	Y	N
	Hastings	5	6		55	83	N	N
	Hastings Degree Operations	10	18		114	160	N	N
	Sin D 5term	1	14		0	0	Y	N
	Sin D 4term	1	13		0	0	Y	N
	Cos D 5term	1	16		0	0	Y	N
	Cos D 4term	1	15		0	0	Y	N
	Tan D 5term	1	12		0	0	Y	N
	Tan D 4term	1	12		0	0	Y	N
	Hastings Radian Operations	12	46		140	216	N	Y
	Cos R 5term	1	16		0	0	Y	Y
	Cos R 4term	1	15		0	0	Y	Y
	Tan R 5term	1	12		0	0	Y	Y
	Tan R 4term	1	12		0	0	Y	Y
	Arctan R 5term	1	18		0	0	Y	Y
	Arctan R 7term	1	17		0	0	Y	N

TABLE A-3. CAMP PARTS SIZING LIST (12 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Comment Size			Part	11th Use
		Spec	Body	Text	Spec	Body			
	Arctan R 6term	1	16		0	0	Y	Y	
	Mod Arctan R 8term	1	28		0	0	Y	N	
	Mod Arctan R 7term	1	27		0	0	Y	N	
	Mod Arctan R 6term	1	26		0	0	Y	N	
	Sin R 5term	1	14		0	0	Y	Y	
	Sin R 4term	1	13		0	0	Y	Y	
	Chebyshev	5	7		59	78	N	N	
	Chebyshev Radian Operations	10	10		103	125	N	N	
	Sin R 5term	1	26		0	0	Y	N	
	Chebyshev Degree Operations	9	11		91	126	N	N	
	Sin D 5term	1	26		0	0	Y	N	
	Chebyshev Semicircle Operations	9	10		91	125	N	N	
	Sin S 5term	1	26		0	0	Y	N	
SUBTOTALS		424	2,893	4,813	3,073	6,513	133	18	
P691	Abstract Data Structures	8	11	3,344	185	105	N	Y	
	Bounded Stack	21	9		138	133	Y	N	
	Clear_Stack	1	5		0	93	N	N	
	Add_Element	2	12		0	137	N	N	
	Retrieve_Element	2	12		0	139	N	N	
	Peek	1	10		0	138	N	N	
	Stack_Status	1	14		0	125	N	N	
	Stack_Length	1	5		0	93	N	N	
	Unbounded Stack	25	10		152	138	Y	N	
	Initialize	1	16		0	124	N	N	
	Clear_Stack	1	18		0	142	N	N	
	Free_Memory	1	13		0	101	N	N	
	Add_Element	2	16		0	148	N	N	
	Retrieve_Element	2	18		0	150	N	N	
	Peek	1	12		0	132	N	N	
	Stack_Status	1	14		0	113	N	N	
	Stack_Length	1	9		0	108	N	N	
	Dot_Next	1	5		0	87	N	N	
	Set_Next	2	6		0	89	N	N	
	Unbounded FIFO Buffer	25	52		160	253	Y	N	
	Initialize_Buffer	1	16		0	111	N	N	
	Clear_Buffer	1	17		0	126	N	N	
	Free_Memory	1	12		0	132	N	N	
	Add_Element	2	16		0	144	N	N	
	Retrieve_Element	2	18		0	156	N	N	
	Peek	1	12		0	124	N	N	
	Buffer_Status	1	14		0	106	N	N	
	Buffer_Length	1	9		0	108	N	N	
	Dot_Next	1	5		0	87	N	N	
	Set_Next	2	6		0	89	N	N	
	Nonblocking Circular Buffer	20	9		147	144	Y	Y	
	Clear_Buffer	1	10		0	100	N	Y	
	Add_Element	2	26		0	125	N	Y	
	Retrieve_Element	2	19		0	137	N	Y	
	Peek	1	17		0	145	N	Y	
	Buffer_Status	1	14		0	123	N	Y	
	Buffer_Length	1	5		0	92	N	Y	
	Unbounded Priority Queue	28	52		174	252	Y	Y	
	Queue_Status	1	14		0	106	N	Y	
	Queue_Length	1	9		0	113	N	Y	
	Dot_Next	1	5		0	87	N	Y	
	Set_Next	2	6		0	89	N	Y	
	Initialize	1	16		0	117	N	Y	
	Clear_Queue	1	18		0	144	N	Y	
	Free_Memory	1	12		0	115	N	Y	
	Add_Element	3	29		0	172	N	Y	
	Retrieve_Element	2	18		0	143	N	Y	
	Peek	1	12		0	121	N	Y	
	Bounded FIFO Buffer	21	9		168	145	Y	Y	
	Peek	1	18		0	141	N	Y	
	Buffer_Status	1	15		0	124	N	Y	
	Buffer_Length	1	5		0	92	N	Y	
	Clear_Buffer	1	10		0	99	N	Y	
	Add_Element	2	19		0	143	N	Y	

TABLE A-3. CAMP PARTS SIZING LIST (13 OF 14)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Comment Size			Part	11th Use
		Spec	Body	Test	Spec	Body			
	Retrieve Element	2	19		0	136		N	Y
	Available Space List Operations	0	6		0	133		N	Y
	New Node	0	17		0	138		N	Y
	Save Node	0	10		0	113		N	Y
	Save List	0	12		0	116		N	Y
SUBTOTALS		204	812	3,344	939	7,331		6	29
P692	Abstract Processes	2	0	0	0	0		Y	M
	Finite State Machine	8	0		0	0		N	N
	Mealy Machine	8	0		0	0		N	N
	Event-Driven Sequencer	8	0		0	0		N	N
	Time-Driven Sequencer	2	0		0	0		N	N
	Sequence Controller	3	0		0	0		N	N
SUBTOTALS		29	0	0	0	0		0	0
P851	Unit Conversions	144	216	961	181	175		N	Y
	Kilograms per Meter Squared and Pounds per Foot Squared	5	2		0	0		N	N
	Conversion to Pounds per Foot2	3	7		0	0		Y	N
	Conversion to Kilograms per Meter2	3	7		0	0		Y	N
	Radians and Semicircles per Second	5	2		0	0		N	N
	Conversion to Semicircles per Second	2	4		0	0		Y	N
	Conversion to Radians per Second	2	4		0	0		Y	N
	Degrees and Semicircles	5	2		0	0		N	N
	Conversion to Semicircles	1	4		0	0		Y	N
	Conversion to Degrees	1	4		0	0		Y	N
	Degrees and Semicircles per Second	5	2		0	0		N	N
	Conversion to Semicircles per Second	3	6		0	0		Y	N
	Conversion to Degrees per Second	3	6		0	0		Y	N
	Seconds and Minutes	5	2		0	0		N	N
	Conversion to Minutes	1	4		0	0		Y	N
	Conversion to Seconds	1	4		0	0		Y	N
	Centigrade and Fahrenheit	5	2		0	0		N	N
	Conversion to Fahrenheit	2	6		0	0		Y	N
	Conversion to Centigrade	2	6		0	0		Y	N
	Centigrade and Kelvin	5	2		0	0		N	N
	Conversion to Kelvin	1	4		0	0		Y	N
	Conversion to Centigrade	1	4		0	0		Y	N
	Fahrenheit and Kelvin	5	2		0	0		N	N
	Conversion to Kelvin	1	5		0	0		Y	N
	Conversion to Fahrenheit	1	5		0	0		Y	N
	Kilograms and Pounds	5	2		0	0		N	N
	Conversion to Kilograms	1	4		0	0		Y	N
	Conversion to Pounds	1	4		0	0		Y	N
	Meters and Feet per Second	5	2		0	0		N	N
	Conversion to Feet per Second	2	5		0	0		Y	N
	Conversion to Meters per Second	2	5		0	0		Y	Y
	Meters and Feet per Second Squared	5	2		0	0		N	N
	Conversion to Feet per Second2	2	6		0	0		Y	N
	Conversion to Meters per Second2	2	6		0	0		Y	N
	Gees and Meters per Second Squared	5	2		0	0		N	N
	Conversion to Meters per Second2	2	5		0	0		Y	Y
	Conversion to Gees	2	5		0	0		Y	N
	Gees and Feet per Second Squared	5	2		0	0		N	N
	Conversion to Feet per Second2	2	5		0	0		Y	N
	Conversion to Gees	2	5		0	0		Y	N
	Radians and Degrees	5	2		0	0		N	N
	Conversion to Degrees	1	4		0	0		Y	N
	Conversion to Radians	1	4		0	0		Y	N
	Radians and Degrees per Second	5	2		0	0		N	N
	Conversion to Degrees per Second	2	6		0	0		Y	N
	Conversion to Radians per Second	2	6		0	0		Y	N
	Radians and Semicircles	5	2		0	0		N	Y
	Conversion to Semicircles	1	5		0	0		Y	Y
	Conversion to Radians	1	5		0	0		Y	Y

TABLE A-3. CAMP PARTS SIZING LIST (CONCLUDED)

TLCSC No.	TLCSC Name Lower Level Units	Code Size			Comment Size		Part	11th Use
		Spec	Body	Test	Spec	Body		
	Meters and Feet	5	2		0	0	N	N
	Conversion to Feet	1	4		0	0	Y	N
	Conversion to Meters	1	4		0	0	Y	N
SUBTOTALS		141	202	961	0	0	34	5
P852	External Form Conversion Twos Complement	35	20	241	236	462	N	Y
	Scale	2	9		0	145	Y	N
	Unscale	2			0	154	Y	Y
SUBTOTALS		4	17	241	0	299	2	1
P890	Quaternion Operations	20	9	194	163	72	N	Y
	Quaternion Computed From Euler Angles	15	26		128	198	Y	Y
	Normalized Quaternion	3	18		65	151	Y	N
	"q"	4	24		65	126	Y	Y
SUBTOTALS		22	68	194	258	475	3	2
TOTALS		5,196	11,768	29,045	29,887	65,532	433	173
CODE TOTALS			16,964	29,045		95,419		
GRAND TOTAL						141,428		

3. DATA BASE ISSUES

Due to the definition and nature of the parts, some difficulties arose concerning the storage of information about parts in the data base. Parts may be TLCSCs, LLCSCs, or units. This means that counting the code for each part can become problematical because a part is not synonymous with an Ada structure. For example, a package may contain three parts. Obviously the specification and body for each part are counted with the part, but what about code for the encapsulating package. Can that be allocated to each part in the same way? The problem was solved by representing each Ada structure in the data base, whether part or encapsulating structure, and designating whether or not an entry was a part. This allows maximum flexibility as to parts designation while at the same time allowing all the Ada code to be represented and counted in the data base.

Another difficulty which arose concerned the hierarchical nature of the parts. Because the parts are implemented as a collection of TLCSCs, and the TLCSCs are packages in Ada, the parts are expressed as a hierarchy of packages and units. In order for the parts to be represented in the data base, this hierarchy must be represented in some way. This may be done in a relational data base, but it is somewhat awkward. ORACLE provides a way for a hierarchy to be expressed, but in order to do so, the parent unit of each part needed to be recorded in the data base. This awkwardness made the generation of reports more difficult and less flexible.

Because no single field could uniquely identify an entry in the relations, surrogates were used. A surrogate is an arbitrary field, usually a number, which is used as the prime key in a data base. The *partno* column in each of the relations contained this surrogate number. The surrogate number was also used to identify an entry's parent. Because the surrogates enabled entries to be identified uniquely by the use of only one field, the hierarchy structure was considerably simpler than if more than one field had to be used as a prime key. The relations were also indexed to provide more speed in referencing.

4. CONCLUSIONS AND RECOMMENDATIONS

The use of the data base enhanced CAMP report capabilities in several areas. The first was the amount of time spent on the reports, particularly editing and formatting. The time spent editing and reformatting the reports must be balanced against the time spent learning the particular data base used and then designing the data base. This learning time, however, was a one-time expenditure while formatting and editing tasks were repeated over and over.

Report availability was also greatly enhanced by the use of the data base. Before the data base method was used, up-to-date reports were not often available and out-of-date reports were used because of the time required to redo the report. With the data base method, a new up-to-date report could be generated very quickly simply by running the report program. In addition, updates were very easy and could be made as they occurred, rather than waiting to have enough to justify spending the time to reformat the entire report.

The number of reports was increased by using the data base. Because the report generating language had to be learned only once, additional reports took only a fraction of the time to write than the initial report.

The utilities and correlative programs also made the use of the ORACLE data base productive. ORACLE has a full range of associated programs available with it which are extremely helpful. In particular, SQL*Forms made the data base interface particularly easy and productive to use. Time spent on data entry was considerably reduced and new people were able to use the interface with minimal instruction (less than 1 hour).

SQL*Plus, the data definition and manipulation language, also made the use of the data base productive. SQL*Plus, based on the standard SQL language, is a very rich, yet relatively easy to use, product. Its use made many data base tasks such as the relation definition easy. Again, productivity needs to be measured against the time spent learning the language, but SQL is relatively standard and can be learned relatively easily by a novice and very easily by anyone with experience with other relational data base query languages. On the other hand, SQL*Plus has a full range of capabilities which can satisfy even the most complex relational application requirements.

The use of a data base for these types of report and information storage needs is highly recommended. A number of lessons concerning the use of the data base came to light during the CAMP usage.

- Data base design should take its place with other software design tasks from the beginning of the project. On CAMP, the use of the data base began after the project was under way. Because of this, there was a duplication of effort when the change was made from using a hand-edited report to a data base. To avoid this type of duplication of effort, it is recommended that a project start with the data base from the beginning.
- Careful attention is required during the initial design and layout phase. The nature and extent of the data already collected when data base use began constrained this phase during CAMP. As a result, the first set of data base relations were designed with little knowledge of how they might need to be expanded or used at a later date with other data base relations. This resulted in less flexibility and more difficult generation of reports later on. Careful data base design at the beginning of the project will reward the extra time spent with fewer problems later on.
- The use of ORACLE is recommended for this type of data base use. The CAMP project found ORACLE easy to use, extremely powerful, and with an excellent set of utilities and correlative programs. ORACLE has the added advantage of using SQL, which is as close to a standard as exists for query languages, and is available on a wide range of equipment.

APPENDIX B

CATALOG ATTRIBUTES

A detailed explanation of each attribute of the CAMP software parts catalog is presented here. For each attribute the following information is provided (as applicable):

1. The *name* of the attribute.
2. The *data type* of the attribute. The type of an attribute can be **NUMERIC** (e.g., *Part Number* is a numeric attribute), **STRING** (e.g., *Part Name* is of type string), **SET** (e.g., the *Withs* attribute may have a set of one or more values), **TEXT** (e.g., the value of *Abstract* is of type text), or **ENUMERATION** (e.g., the *Mode* attribute must have a value of *bundled*, *unbundled*, or *schematic*).
3. The *domain* of an **ENUMERATION** type.
4. The *status* of the attribute. This is either **REQUIRED** (i.e., all parts must be supplied a value for this attribute) or **RECOMMENDED** (i.e., the attribute is recommended for completeness but not required).
5. A *description* of the attribute's meaning, including mention of any default values and the source (user or system) of attribute entry.
6. An *example* of a valid value is shown for each attribute.

The catalog attributes are enumerated in Figure B-1.

GENERAL

Part Number	Revision Number
Part Name	Functional Abstract
Mode	Taxonomic Category
Class	Keywords
Last Change Date of Entry	Project Usage
Government Security Classification (part)	Corporate Sensitivity Level (part)
Government Security Classification (entry)	Corporate Sensitivity Level (entry)
Remarks	

DEVELOPMENT

Design Issues	Revision Notes
Development Date	Developer
Development Status	Developed For
Requirements Documentation	Design Documentation

USAGE

Location of Source Code	Access Notes
Withs	Withed By
Implemented By	Implements
Built From	Used to Build
Sample Usage	Hardware Dependencies
Restrictions	

PERFORMANCE

Source Size/Complexity Characterizations	Fixed Object Code Size
Timing	Accuracy

Figure B-1. Catalog Attributes

ATTRIBUTE NAMEPart Number
TYPENumeric
STATUSRequired
DESCRIPTIONPart Number is an integer which together with the value of the Revision Number attribute uniquely identifies a catalog entry. The Part Number is not required to be unique (i.e., the same number would be used for all revisions of a given part). The Part ID code will consist of the letter P followed by the Part Number hyphenated with the Revision Number, and will be generated by the system. The part number should not contain leading zeroes.
EXAMPLE16

ATTRIBUTE NAMERevision Number
TYPENumeric
STATUSRequired
DESCRIPTIONThe Revision Number is an integer used to uniquely identify revisions of a particular part. The revision number will be generated by the system. The first entry will always to be 0, with subsequent revision values incrementing by 1. This value together with the Part Number form a unique key called the Part ID.
EXAMPLE5

ATTRIBUTE NAMEPart Name
TYPEString
STATUSRequired
DESCRIPTIONA valid Ada identifier which provides a brief, and not necessarily unique, descriptive name for a part (e.g., a package may have more than one body, in which case both bodies would have the same name but they would be uniquely identifiable by the Part ID).
EXAMPLEMissile_Launch_Platform

ATTRIBUTE NAMEGovernment Security Classification of Part
TYPEEnumeration
DOMAIN(Unclassified, Confidential, Secret, Top_Secret)
STATUSRequired
DESCRIPTIONSpecifies the DoD security classification of the part. The default value is Unclassified.
EXAMPLEUnclassified

ATTRIBUTE NAMECorporate Sensitivity Level of Part
TYPEEnumeration
DOMAIN(Unclassified, Private, Sensitive, Proprietary)
STATUSRequired
DESCRIPTIONSpecifies the corporate sensitivity level of the part. The default value is Unclassified.
EXAMPLESensitive

ATTRIBUTE NAMEGovernment Security Classification of Catalog Entry

TYPEEnumeration

DOMAIN(Unclassified, Confidential, Secret, Top_Secret)

STATUS.....Required

DESCRIPTIONSpecifies the DoD security classification of a part's catalog entry; this may be different from the security classification of the part itself. The default value is Unclassified.

EXAMPLE.....Secret

ATTRIBUTE NAMECorporate Sensitivity Level of Catalog Entry

TYPEEnumeration

DOMAIN(Unclassified, Private, Sensitive, Proprietary)

STATUS.....Required

DESCRIPTIONSpecifies the corporate sensitivity level of a part's catalog entry; this may be different from the corporate sensitivity level of the part itself. The default value is Unclassified.

EXAMPLE.....Proprietary

ATTRIBUTE NAMETaxonomic Category

TYPEConcatenation of enumeration values

DOMAINsee Figure B-2

STATUS.....Required

DESCRIPTIONSpecifies the taxonomic classification of the part. This can be a multi-leveled specification, using periods to separate the different levels of classification.

EXAMPLE.....Primary Operation.Navigation

ATTRIBUTE NAMEFunctional Abstract

TYPEText

STATUS.....Required

DESCRIPTIONA brief (no greater than 500 words) explanation of the purpose and functionality of the part. This attribute is intended to provide the user with a quick overview of the unit.

EXAMPLE.....The bounded FIFO buffer performs buffering of data in a first-in first-out fashion. The part will limit the number of items which may be in the buffer at any one time and will raise an exception if an attempt is made to add to an already full buffer. The part can be used to buffer incoming Mission Data, TERCOM processing, or DSMAC updates. In addition, this part can be used for message passing between components of a system.

ATTRIBUTE NAMEDesign Issues

TYPEText

STATUS.....Recommended

DESCRIPTIONThis attribute should briefly discuss the rationale for design decisions such as the selection of data structures and algorithms to be used. The user should be referred to external design documentation for a lengthy discussion of the issues. This field should contain information on the use of pragma inline for the part under consideration.

EXAMPLE.....Since the telemetry sampling rate changes depending upon the values of the input data, the quantity of data to be buffered is impossible to know in advance. For this reason, dynamic buffers have been used for telemetry data storage buffering.

ATTRIBUTE NAME **Revision Notes**
TYPE **Text**
STATUS **Recommended**
DESCRIPTION This attribute should briefly describe the reason for revision, and any changes in functionality that have occurred as a result of the revision.
EXAMPLE The matrix multiply of the H and J matrices was changed. A diagonal matrix multiply routine is now utilized rather than the more general matrix multiply routine previously used. This was found to be appropriate for every case and the change does not affect functionality, but results in a more efficient part.

ATTRIBUTE NAME **Class**
TYPE **Enumeration**
DOMAIN (Package Specification, Package Body, Task Specification, Task Body, Subprogram Specification, Subprogram Body, Generic Package Specification, Generic Package Body, Generic Task Specification, Generic Task Body, Generic Subprogram Specification, Generic Subprogram Body, Generic Formal Part, Context Clause)
STATUS **Required**
DESCRIPTION This attribute specifies the type of the part. The word *type* is not used for this attribute in order to avoid confusion with Ada types.
EXAMPLE Task Body

ATTRIBUTE NAME **Keywords**
TYPE Set of 0 or more Strings
STATUS **Recommended**
DESCRIPTION This attribute contains one or more keywords or phrases that can be used to locate a part. Keywords narrow the search for a desired part. Keywords can be used to describe functionality of the part, or task area. Keywords are entered for the top-level specification only, although they apply to the lower levels as well.
EXAMPLE (autopilot, navigation)

ATTRIBUTE NAME **Mode**
TYPE **Enumeration**
DOMAIN (Bundled Code, Unbundled Code, Schematic)
STATUS **Required**
DESCRIPTION This attribute indicates the part's usage mode. Bundled parts come complete with an *environment*. Unbundled parts consist of the part itself; the user must establish the environment in which it is to be used. Schematic parts must be constructed from the *constructors* provided.
EXAMPLE bundled code

ATTRIBUTE NAME **Last Change Date of Entry**
TYPE **String**
STATUS **Required**
DESCRIPTION This attribute provides the date that the catalog entry was last changed; it will be supplied by the system.
EXAMPLE 07-30-85

ATTRIBUTE NAMEDevelopment Date

TYPEString

STATUS.....Required

DESCRIPTIONThis attribute provides the date that the original part or revision was developed; it will be supplied by the user.

EXAMPLE.....02-22-85

ATTRIBUTE NAMEDeveloper

TYPEString

STATUS.....Required

DESCRIPTIONThis entry identifies the name of the organization that developed the part. The default is McDonnell Douglas Astronautics Co.

EXAMPLE.....McDonnell Douglas Astronautics Co.

ATTRIBUTE NAMEDeveloped For

TYPESet of strings

STATUS.....Recommended

DESCRIPTIONThis attribute should identify the project and type of software for which the part was originally developed. Multiple entries are allowed for this attribute.

EXAMPLE.....Tomahawk (BGM-109AS) Flight Software

ATTRIBUTE NAMEDevelopment Status

TYPEEnumeration

DOMAIN.....(Specified, Designed & Coded, Tested, Verified)

STATUS.....Required

DESCRIPTIONThis attribute indicates the development status of the unit. If the value is *Specified*, this indicates that the need for and purpose of the part have been identified and the requirements have been specified (all required attributes except for Mode, Withs, and Withed By should be supplied for a part in this state). If *Designed & Coded*, the requirements for the part have been refined and used to specify the part for coding in Ada so that compiled code is now available (all remaining attributes may now be supplied). A part with development status of *Tested* indicates that this part has passed the tests of the developer and found to be in working condition. Status of *Verified* indicates that the part has been accepted and verified by the customer for which it was originally developed.

EXAMPLE.....Tested

ATTRIBUTE NAMESource Size/Complexity Characterizations

TYPEText

STATUS.....Recommended

DESCRIPTIONThis attribute provides the size of the Ada part in terms of lines of source code (LOC), and other complexity characterizations.

EXAMPLE.....Lines of Source Code:

15 lines executable
2 lines type declarations
5 lines object declarations

ATTRIBUTE NAMEFixed Object Code Size

TYPEText

STATUS.....Recommended

DESCRIPTIONThis attribute provides the fixed (static) size of the Ada part in terms of bytes of object code. It is environment-dependent, therefore, the conditions under which the figure was obtained must be provided.

EXAMPLE.....720 bytes when compiled on VAX 11/780 using the VAX Ada compiler.

ATTRIBUTE NAMELocation of Source Code

TYPEString

STATUS.....Recommended

DESCRIPTIONThis entry should specify the file name, library, and computer system where the source code for the part or part constructor is located. A value for this attribute is entered for the top-level specification only, although it applies to the lower levels as well.

EXAMPLE.....USERDISK5:[CAMP2.ABSTRACT]FSM.ADA

ATTRIBUTE NAMEAccess Notes

TYPEText

STATUS.....Recommended

DESCRIPTIONThis attribute provides access information for a particular part. To deal with actual Ada parts, information is given to aid in applying the Ada compilation rules for part use, such as what other parts must be withed. For schematic parts, information is given on how to get to a particular part, such as how to invoke the schematic constructor.

EXAMPLE.....Include the statement "with Matrix_Types".

ATTRIBUTE NAMERequirements Documentation

TYPEText

STATUS.....Recommended

DESCRIPTIONThis attribute identifies the requirements documentation and indicates its availability.

EXAMPLE.....Cruise Missile Advanced Guidance Computer Program Development Specification for the Inertial Navigation System, Specification #70H541092

ATTRIBUTE NAMEDesign Documentation

TYPEText

STATUS.....Recommended

DESCRIPTIONThis attribute identifies the design documentation and indicates its availability.

EXAMPLE.....Software Detailed Design Document for the Missile Software Parts of the Common Ada Missile Packages Project

ATTRIBUTE NAMEWiths

TYPESet of identifiers composed of Part IDs

STATUS.....Required

DESCRIPTIONThis attribute contains an enumeration of other units within the catalog that this unit withs.

EXAMPLE.....(P160-2, P161-2)

ATTRIBUTE NAME Withed By
TYPE Set of identifiers composed of Part IDs
STATUS Required
DESCRIPTION This attribute contains an enumeration of other units within the catalog that *with* this unit.
EXAMPLE P70-0

ATTRIBUTE NAME Project Usage
TYPE Set of strings
STATUS Recommended
DESCRIPTION This attribute enumerates the projects and systems that use this particular part. The places where components generated via constructors are used should also be enumerated. The usage attribute aids in tracking systems which have 'checked a part out of the library'. Such an entry facilitates maintenance in the event that an error is found in a part.
EXAMPLE (AGM-109H, AGM-109L, Harpoon)

ATTRIBUTE NAME Sample Usage
TYPE Text
STATUS Recommended
DESCRIPTION This attribute provides the user with the information necessary to use the part (i.e., how, when, and where the part should be used). Potential usage of this part in the applications of an organization may be discussed here.
EXAMPLE This part is generally a candidate for use in any missile which has a throttleable engine and which requires the control of mach number.

ATTRIBUTE NAME Accuracy
TYPE Text
STATUS Recommended
DESCRIPTION This field contains information on the algorithmic accuracy or precision of numerical results computed by the part. If this information is not relevant, it should be left blank.
EXAMPLE The distance returned has an accuracy of 15 significant digits.

ATTRIBUTE NAME Timing
TYPE Text
STATUS Recommended
DESCRIPTION This field contains information on execution time for sample invocations or instantiations of the part. The run-time conditions that produced the timing results must be specified in order to make this information relevant.
EXAMPLE This part averaged an execution time of 0.52 milliseconds when called 200 times from a continuous loop on a dedicated Microvax II.

ATTRIBUTE NAME Implements
TYPE Set of identifiers composed of Part IDs
STATUS Recommended
DESCRIPTION This attribute is valid only for a *body*, and identifies the specification portion that it implements.
EXAMPLE P603-5

ATTRIBUTE NAME **Implemented By**
TYPE Set of identifiers composed of Part IDs
STATUS Recommended
DESCRIPTION This attribute is valid only for a *Specification*, and identifies the body or bodies that implement it.
EXAMPLE P603-5

ATTRIBUTE NAME **Built From**
TYPE Set of identifiers composed of Part IDs
STATUS Recommended
DESCRIPTION This attribute consists of an enumeration of other units within the catalog which are encapsulated within this unit; these are the parts which this unit is built from. The entries must be the Part IDs of these parts. Table B-1 provides guidelines for determining possible *built from* relationships for parts.
EXAMPLE P603-5

ATTRIBUTE NAME **Used to Build**
TYPE Set of identifiers composed of Part IDs
STATUS Recommended
DESCRIPTION This attribute consists of an enumeration of other units within the catalog which encapsulate this unit; these are the parts which are used to build the current part. The entries must be the Part IDs of these parts. Table B-1 provides guidelines for determining possible *used to build* relationships for parts.
EXAMPLE P603-5

ATTRIBUTE NAME **Hardware Dependencies**
TYPE Text
STATUS Recommended
DESCRIPTION This entry contains an elaboration of any hardware dependencies of the part which would limit its transportability. If there are no dependencies, this attribute will show *None*.
EXAMPLE 1553B data bus

ATTRIBUTE NAME **Restrictions**
TYPE Text
STATUS Recommended
DESCRIPTION This attribute indicates any usage restrictions such as proprietary rights and copyrights.
EXAMPLE This part is not to be used without the express written permission of McDonnell Douglas Astronautics Company.

ATTRIBUTE NAME **Remarks**
TYPE Text
STATUS Recommended
DESCRIPTION This field is for any general remarks concerning the part, or for continuations of other fields.
EXAMPLE It is anticipated that future missiles will use the structures contained in this part to control external message handling and to support dynamic task priorities in Ada.

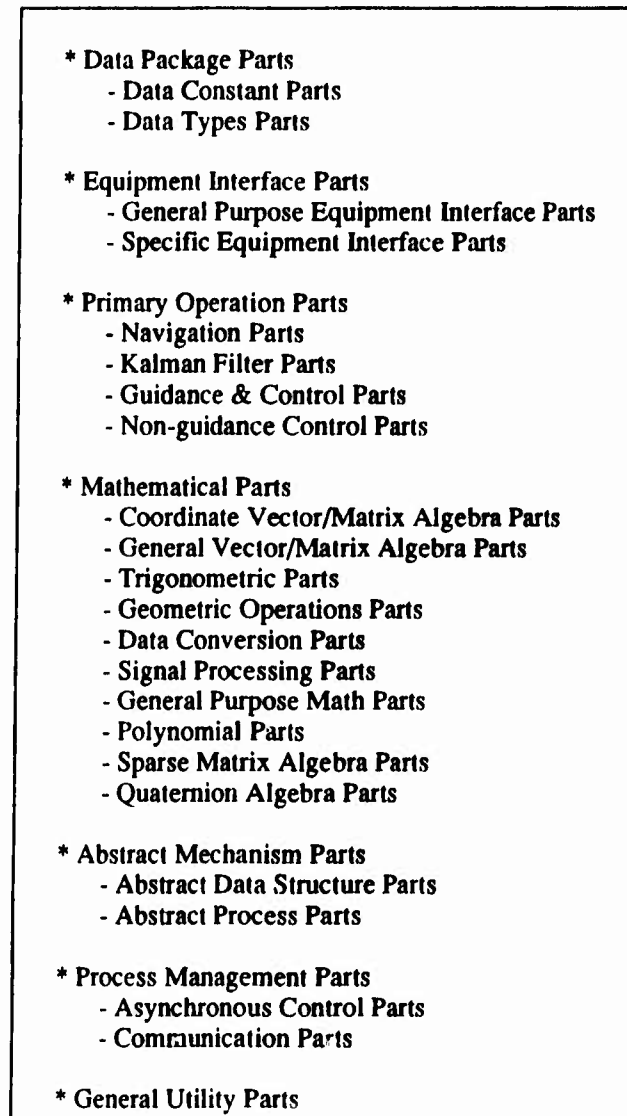


Figure B-2. CAMP Parts Taxonomy

TABLE B-1. 'USED TO BUILD' AND 'BUILT FROM' ATTRIBUTE RELATIONSHIPS

Part Class	Used to Build	Built From
Package Specification	Package Specification Subprogram Body Package Body Task Body	Package Specification Subprogram Specification Task Specification
Package Body	Package Specification	Package Specification Subprogram Specification Task Specification
Subprogram Specification	Package Specification Package Body Task Body	
Subprogram Body	Subprogram Specification	Package Specification Subprogram Specification Task Specification
Task Specification	Package Specification Subprogram Body Package Body	Task Body

References

- [1] M.R. Barbacci, A.N. Habermann, and M. Shaw, "The Software Engineering Institute: Bridging Practice and Potential", *IEEE Software*, November 1985.
- [2] STARS, "Technical Program Plan". Tech. report, STARS Joint Program Office, June 1986.
- [3] DoD, "Management of Computer Resources in Major Defense Systems", DoD Directive (Draft) 5000.29, Office of the Under Secretary of Defense, Computer Software and Systems, April 1986.
- [4] R.D. DeLauer, "Interim DoD Policy on Computer Programming Languages", Letter issued by the Under Secretary of Defense, Research and Engineering.
- [5] DoD, "Computer Programming Language Policy", DoD Directive 3405.1, Department of Defense, April 1987.
- [6] DoD, "Use of Ada in Weapon Systems", DoD Directive 3405.2, Department of Defense, March 1987.
- [7] D.G. McNicholl, C. Palmer, et al, "Common Ada Missile Packages (CAMP), Volume II: Software Parts Composition Study Results", Tech. report AFATL-TR-85-93, Air Force Armament Laboratory, May 1986, (Must be acquired from DTIC using access number B102655. Distribution limited to DoD and DoD contractors only.)
- [8] United States Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983 ed., 1983.
- [9] R.L. Schwartz, P.M. Melliar-Smith, "The Suitability of Ada for Artificial Intelligence Applications", Final Report, SRI International, May 1980.
- [10] Boeing Aerospace Co., "Software Interoperability and Reusability", Tech. report RADC-TR-83-174, Rome Air Development Center, July 1983, Volume I, p. 105
- [11] G. Booch, *Software Engineering with Ada*, Benjamin Cummings, Menlo Park, CA, 1983, pp. 202.
- [12] R. Leavitt, "Some Practical Experience in the Organization of a Library of Reuseable Ada Units", *Proceedings, Third Annual National Conference on Ada Technology*, 1985, pp. 70.
- [13] S.D. Litvinchouk, and A.S. Matsumoto, "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 550.
- [14] T.A. Standish, "An Essay on Software Reuse", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 496.

INITIAL DISTRIBUTION LIST

GTE GOVERNMENT SYS CORP	1	CARNEGIE MELLON UNIV/	
ADVANCED DIGITAL SYSTEMS	1	SOFTWARE ENGINEERING INST	1
AFATL/FXG	4	NOAA/ERL/R/E/AL4	1
MILITARY COMPUTER SYSTEMS	1	INTERMETRICS, INC/G. RENTH	1
LOCKHEED/O/62-81, B/563, F15	1	INTERMETRICS, INC/D.P. SMITH	1
HUGHES/FULLERTON	1	FORD AEROSPACE/WEST DEVEL DIV	1
UNISYS/MS-E1D08	1	AD/ENE	1
WESTINGHOUSE/BALTIMORE	1	ROCKWELL/MS-GA21	1
AFWAL/AAAS-2	1	GRUMMAN CORP/MS D-31-237	1
BOOZ-ALLEN & HAMILTON, INC	1	INSTITUTE OF DEFENSE ANALYSIS	1
BOEING AEROSPACE COMPANY/MS 8H-09	1	TELEDYNE BROWN/MS 178	1
BOEING AEROSPACE CO	1	USAF/TAWC/SCAM	1
AD/YGE	1	BOEING AEROSPACE CO/D. LINDBERG	1
SOFTWARE PRODUCTIVITY CONSORTIUM	5	LOGICON	1
ARMY CECOM/AMSEL-COM-IA	1	EASTMAN KODAK/DEPT 47	1
NAVAL TRAINING SYS CENTER/CODE 251	1	SYSTEMS CONTROL TECH, INC	1
SCIENCE APPLICATIONS INTL CORP	1	E-SYSTEMS/GARLAND DIV	1
RAYTHEON/MSL SYS DIVISION	1	AFWAL/AAAF	1
CALSPAN	1	MARTIN DEVELOPMENT	1
KAMAN SCIENCES CORPORATION	1	MA COMPUTER ASSOCIATES INC	1
NAVAL RESEARCH LAB/CODE 5595	1	IBM FEDERAL SYS DIV/MC 3206C	1
CARNEGIE MELLON UNIV/SEI/SOLOM	1	MCDONNELL DOUGLAS/INCO, INC	1
COLEMAN RESEARCH CORP	1	UNITED TECH, ADVANCED SYS	1
COLSA, INC	1	MCDONNELL AIRCRAFT CO/DEPT 300	1
CONTROL DATA CORPORATION	1	WESTINGHOUSE ELEC/MS 432	1
WINTEC	1	MHP FU-TECH, INC	1
CONTROL DATA/DEPT 1855	1	ITT AVIONICS	1
DACS/RADC/COED	1	COSMIC/UNIV OF GA	1
RAYTHEON/EQPT DIV	1	NAVAL OCEAN SYS CENTER/CODE 423	1
BMO/ACD	1	NAVAL WEAPONS CTR/CODE 3922	1
DDC-I, INC	1	ODYSSEY RESEARCH ASSOCIATES, INC	1
ENGINEERING & ECONOMICS RESEARCH/		USA ELEC PROVING GRD/STEEP MT-DA	1
DIV OFFICE	1	PATHFINDER SYS	1
BDM CORP	1	BDM CORPORATION	1
AFATL/FXG/EVERS	1	PERCEPTRONICS, INC	1
ESD/SYW-JPMO	1	PHOENIX INTERNATIONAL	1
FORD AEROSPACE & COMM CORP/MS H04	1	MCDONNELL DOUGLAS ASTRO CO	1
UNIV OF COLORADO #202	1	GTE LABORATORY/RUBEN PRIETO-DIAZ	1
ANALYTICS	1	PROPRIETARY SOFTWARE SYSTEMS	1
AFWAL/FIGL	1	ADVANCED TECHNOLOGY	8
WESTINGHOUSE ELECTRIC CORP/MS 5220	1	STANFORD TELECOMMUNICATIONS, INC	1
GENERAL DYNAMICS/MZ W2-5530	1	RATIONAL	1
HONEYWELL INC	1	LOCKHEED MISSILES & SPACE CO	1
TAMSCO	1	HERCULES DEFENSE ELEC SYS	1
STARS	1	AEROSPACE CORP	1
FORD AEROSPACE/MS 2/206	1	ROGERS ENGINEERING & ASSOCIATES	1
GRUMMAN HOUSTON CORPORATION	1	ADASOFT INC	1
NAVAL AVIONICS CENTER/NAC-825	1	ESD/XRSE	1
NASA JOHNSON SPACE CENTER/EH/GHG	1	SANDERS/MER 24-1212	1
BOEING AEROSPACE/MS-8Y97	1	CSC/ERIC SCHACHT	1
HARRIS CORPORATION/GISD	1	COMPUTER TECH ASSOCIATES, INC	1

INITIAL DISTRIBUTION LIST (CONCLUDED)

SCIENCE APPLICATIONS INTER CORP	1	FTD/SDNF	1
HQ CASE/CBRC	1	AFWAL/FIES/SURVIAC	1
GOULD INC/CSD	1	HQ USAFE/INATW	1
HQ AFSPACECOM/LKWD/STOP 32	1	AFATL/CC	1
SVERDRUP/EGLIN	1	AFATL/CA	1
HONEYWELL INC/CLEARWATER	1	AFATL/DOIL	2
TECHNOLOGY SERVICE CORP	1	6575 SCHOOL SQUADRON	1
AEROSPACE/LOS ANGELES	1	IITRI	1
SOFTWARE ARCHITECTURE & ENGIN	1		
LORAL SYSTEMS GROUP/D/476-C2E	1		
NADC/CODE 7033	1		
UNISYS/PAOLA RESEARCH CTR	1		
SIRIUS INC	1		
GENERAL RESEARCH CORP	1		
SOFTECH, INC/R.L. ZALKAN	1		
SOFTECH, INC/R.B. QUANRUD	1		
SOFTWARE CERTIFICATION INS	1		
SOFTWARE CONSULTING SPECIALIST	1		
SOFTWARE PRODUCTIVITY SOLUTIONS, INC	1		
STAR-GLO INDUSTRIES INC	1		
NADC/CODE 50C	1		
WESTINGHOUSE/BALTIMORE	1		
MITRE CORPORATION	1		
SYSCON CORP/I. WEBER	1		
SYSCON CORP/C. MORSE	1		
SYSCON CORP/T. GROBICKI	1		
AEROSPACE CORPORATION/M-8-026	1		
TEXTRON DEFENSE SYSTEMS	1		
GENERAL DYNAMICS/MZ 1774	1		
TIBURON SYSTEMS, INC	1		
TRW DEFENSE SYS GROUP	1		
NASA SPACE STATION	1		
BALLISTIC MSL DEF ADVANCED/ TECHNOLOGY CENTER	1		
IBM CORPORATION/FSD	1		
VISTA CONTROLS CORPORATION	1		
VITRO CORPORATION	1		
NAVAL RESEARCH LABORATORY/CODE 5150	1		
CACI, INC	1		
AFSC/PLR	5		
DIRECTOR ADA JOINT PROGRAM OFFICE	1		
MCDONNELL DOUGLAS ASTRONAUTICS/ E 434/106/2/MS22	7		
SDIO/S/PI	1		
ADVANCED SOFTWARE TECH SPECIALTIES	1		
DTIC-DDAC	2		
AFCSA/SAMI	1		
AUL/LSE	1		

SUPPLEMENTARY

INFORMATION



DEPARTMENT OF THE AIR FORCE
WRIGHT LABORATORY (AFSC)
EGLIN AIR FORCE BASE, FLORIDA, 32542-5434



REPLY TO
ATTN OF:

MNOI

ERRATA
AD-8729568

13 Feb 92

SUBJECT: Removal of Distribution Statement and Export-Control Warning Notices

TO: Defense Technical Information Center
ATTN: DTIC/HAR (Mr William Bush)
Bldg 5, Cameron Station
Alexandria, VA 22304-6145

1. The following technical reports have been approved for public release by the local Public Affairs Office (copy attached).

<u>Technical Report Number</u>	<u>AD Number</u>
1. 88-18-Vol-4	ADB 120 251
2. 88-18-Vol-5	ADB 120 252
3. 88-18-Vol-6	ADB 120 253
4. 88-25-Vol-1	ADB 120 309
5. 88-25-Vol-2	ADB 120 310
6. 88-62-Vol-1	ADB 129 568
7. 88-62-Vol-2	ADB 129 569
8. 88-62-Vol-3	ADB 129-570
9. 85-93-Vol-1	ADB 102-654 ✓
10. 85-93-Vol-2	ADB 102-655
11. 85-93-Vol-3	ADB 102-656
12. 88-18-Vol-1	ADB 120 248
13. 88-18-Vol-2	ADB 120 249
14. 88-18-Vol-7	ADB 120 254
15. 88-18-Vol-8	ADB 120 255 ✓
16. 88-18-Vol-9	ADB 120 256
17. 88-18-Vol-10	ADB 120 257 ✗
18. 88-18-Vol-11	ADB 120 258
19. 88-18-Vol-12	ADB 120 259

2. If you have any questions regarding this request call me at DSN 872-4620.

Lynn S. Wargo
LYNN S. WARGO

Chief, Scientific and Technical
Information Branch

1 Atch
AFDTC/PA Ltr, dtd 30 Jan 92

ERRATA



DEPARTMENT OF THE AIR FORCE
HEADQUARTERS AIR FORCE DEVELOPMENT TEST CENTER (AFDC)
EGLIN AIR FORCE BASE, FLORIDA 32542-6000



REPLY TO
ATTN OF: PA (Jim Swinson, 882-3931)

30 January 1992

SUBJECT: Clearance for Public Release

TO: WL/MNA

The following technical reports have been reviewed and are approved for public release: AFATL-TR-88-18 (Volumes 1 & 2), AFATL-TR-88-18 (Volumes 4 thru 12), AFATL-TR-88-25 (Volumes 1 & 2), AFATL-TR-88-62 (Volumes 1 thru 3) and AFATL-TR-85-93 (Volumes 1 thru 3).

Virginia N. Pribyla
VIRGINIA N. PRIBYLA, Lt Col, USAF
Chief of Public Affairs

AFDTC/PA 92-039